The University Of Sheffield.

J Vella

An object-oriented data and query model

This thesis is submitted for the degree of

Doctor of Philosophy

2013

The University Of Sheffield.

Access To Thesis.

- A bound copy of every thesis which is accepted as worthy for a higher degree, must be deposited in the University of Sheffield Library, where it will be made available for borrowing or consultation in accordance with University Regulations.
- All students registering from 2008–09 onwards are also required to submit an electronic copy of their final, approved thesis. Students who registered prior to 2008–09 may also submit electronically, but this is not required.

Author: Joseph Vella Dept: Computer Science

Thesis Title: An object-oriented data and query model Registration No: 080124450

For completion by all students:

Submit in print form only (for deposit in the University Library): ☑

Submit in print form and also upload to the *White Rose eTheses Online* server: In full ☑

Edited eThesis ☐

Please indicate if there are any embargo restrictions on this thesis. Please note that if no boxes are ticked, you will have consented to your thesis being made available without any restrictions.

Embargo details: (complete only if requesting an embargo to either your print and/or eThesis)	Embargo required?		Length of embargo (in years)
Print Thesis	Yes ☐	No ☑	____
eThesis	Yes ☐	No ☑	____

Supervisor: I, the supervisor, agree to the named thesis being made available under the conditions specified above.

Name: Dr Anthony J H Simons Dept: Computer Science

Signed:  Date: 4 April 2013

Student: I, the author, agree to the named thesis being made available under the conditions specified above.

I give permission to the University of Sheffield to reproduce the print thesis in whole or in part in order to supply single copies for the purpose of research or private study for a non-commercial purpose.

I confirm that this thesis is my own work, and where materials owned by a third party have been used copyright clearance has been obtained. I am aware of the University's *Guidance on the Use of Unfair Means* (www.sheffield.ac.uk/lets/design/unfair)

I confirm that all copies of the thesis submitted to the University (including electronic copies on CD/DVD) are identical in content.

Name: Joseph Vella Dept: Computer Science

Signed:  Date: 4 April 2013

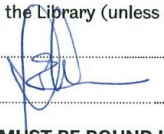
For completion by students also submitting an electronic thesis (eThesis):

I, the author, agree that the University of Sheffield's eThesis repository (currently WREO) will make my eThesis available over the internet via an entirely non-exclusive agreement and that, without changing content, WREO may convert my thesis to any medium or format for the purpose of future preservation and accessibility.

I, the author, agree that the metadata relating to the eThesis will normally appear on both the University's eThesis server and the British Library's EThOS service, even if the thesis is subject to an embargo. I agree that a copy of the eThesis may be supplied to the British Library.

I confirm that the upload is identical to the final, examined and awarded version of the thesis as submitted in print to the University for deposit in the Library (unless edited as indicated above).

Name: Joseph Vella Dept: Computer Science

Signed:  Date: 4 April 2013

THIS SHEET MUST BE BOUND IN THE FRONT OF THE PRINTED THESIS BEFORE IT IS SUBMITTED

An object-oriented data and query model

Joseph Vella

This thesis is submitted for the degree of

Doctor of Philosophy,

in the Department of Computer Science,

University of Sheffield,

March, 2013.

An object-oriented data and query model

Joseph Vella

Abstract

OODBs build effective databases with their development peaking in 2000. A reason given for its neglect is of not having a declarative and procedural language. Relevant declarative languages include ODMG OQL and Flora-2, a first order and object-oriented logic programming system based on F-Logic. Few procedural object algebras have been proposed and ours is one and none are offered together.

The characteristics of the algebra proposed and implemented with Flora-2 are: it is closed; it is typed; its ranges and outputs are homogeneous sets of objects; operators work on either values or logical identifiers; a result set is asserted; and a query expression's meta details are asserted too. The algebra has ten operators and most have algebraic properties.

A framework was developed too and it has its object base loaded with methods that maintain it and our algebra. A framework's function is to read an EERM diagram to assert the respective classes and referential constraints. The framework then sifts for non-implementable constructs in the EERM diagram and converts them into implementable ones (e.g. *n*-ary relationships) and translate the object base design into ODMG ODLs. This translation's correctness and completeness is studied.

The framework implements run-time type checking as Flora-2 lacks these. We develop type checking for methods that are static, with single arity, polymorphic (e.g. overloaded and bounded), and recursive structures (e.g. lists) through well-known and accepted techniques.

A procedure that converts a subset of OQL into an algebraic expression is given. Once created it is manipulated to produce an optimised expression through a number of query re-writing methods: e.g. semantic, join reduction, and view materialisation. These techniques are aided by the underlying object base constructs; e.g. primary key constraint presence is used to avoid duplicate elimination of a result set. We show the importance of tight coupling in each translation step from an EERM to an algebraic expression. Also we identify invariant constructs, e.g. primary key through a select operand, which do not change from a query range to a query result set.

Dedication:

To our Kim Jamie and Kyle Matthew

Acknowledgements:

I am deeply indebted to my supervisors: Anthony JH Simons, Vitezlav Nezval, and Georg Struth. Their support and guidance is much appreciated. Also I have to say how good it feels to be part of the department at the University of Sheffield. Additionally I need to acknowledge and whole-heartedly thank the examiners for viewpoints and suggestions given.

A great thank you goes to Michael Kifer for patiently guiding me, if not illuminating, to aspects of F-Logic and Flora-2.

A thank you goes to the EyeDB community for their occasional help.

I have to say thanks to Dropbox too; it is a life saver.

My employer provided financial support and a two year studies leave; thanks very much.

Special hug and kisses goes to Adriana; you are the one that I love!

Table of Contents

Chapter 1

Introduction

1 – Introduction

This thesis is about data modelling and databases. *Data modelling* deals with the processes and the artefacts required to accurately and consistently describe the data requirements of an application [IBMAC93]. A *database* is a collection of interrelated or independent data items that are stored together to serve an application's data requirements [IBMAC93]. The structure of a collection of data is the result of a data modelling process. Databases are managed by a program called a *Database Management System* (DBMS). The DBMS handling of operations over its data are expected to yield significant benefits in terms of data availability, of correct sharing, data consistency maintenance, query optimisation, and economy of scale for the end users.

Since the Eighties the dominant data modelling paradigm has been the relational. During in the Nineties a richer object-oriented based data modelling came into prominence, but faded after; nonetheless object-oriented programming remains till today main stream. During the Noughties the big data and NoSQL movements, at least from a visibility aspect, become prevalent especially where only two of the targets set for consistency, availability and data partitioning can be met (i.e. Brewer's CAP theorem). In parallel with the relational model, the logical data model was developed where first order semantics was coupled with logic programming; these are called deductive databases.

Within this context it is important to underline two issues. Firstly, within DBMS capabilities there is a trade-of between the expressibility of the data modelling and the potential for query optimisation. Secondly, specifications tend to lose detail when they are mapped from conceptual to logical data schemas, something we are to address.

We are motivated in applying an object-oriented data modelling language that captures a wider portion of an application's data requirements when compared to the relational model or the NoSQL models. It is well known that the wider this portion is the more requirements the DBMS can manage. We are also motivated in using a first-order semantics logic based data model that can encode and infer on object-oriented data modelling. Consequently with one language we can describe the database structures, the

queries over it, define the integrity constrains over the database, the procedures over the database, and even the actual data.

Another part of our motivation is to ensure that the translations required, for example from the data requirements to the conceptual model, and from the conceptual to the logical model, are seamlessly integrated, rule based, and insofar as this is possible, all information encoded in specifications is preserved in every translation. An advantage of this is that specifications encoded remain invariant through translations and therefore it is available for latter phases. For example the primary key concept of an entity, first encoded in the conceptual design, is found and used to optimise and execute a query modelling instance. To address low attrition requirement we are adopting object-oriented friendly features in each translation-target language.

Inspired by these motivations a number of research questions are addressed in this work. The following are a selection of these:

Firstly, which conceptual modelling representation has many of its constructs easily and consistently converted into a logical data model?

Second, which mix of the many object-oriented themes and variations can be selected to form the logical data modelling language used to build a database?

Third, what needs to be included in an object-oriented query model that has an object-oriented data model as its basis?

Fourth, are the data and query models developed here adequate to build complex logical schemas?

Fifth, can the queries over the object database be specified in declarative and procedural languages? And how do these align to known efforts, for example with ODMG OQL?

Sixth, how effective are the tools and standards used in this research, e.g. EERM, ODMG, EyeDB and F-logic, at achieving the stated goals of describing and translating data and query modelling faithfully?

1.1 Thesis Statement

During the development of this thesis write-up we show:

- Our synthesised object-oriented data and query model is still relevant today and is adequate to define a complex database. A justification of its relevance, when for example compared with NoSQL systems can forgo database consistency, or adopt eventual consistency, in favour of data availability and partitioning.
- The cohesion of the data and the query model through object-oriented themes not only makes each model stronger but also enables DBMS led optimisations and more opportunities for automated-software development in certain database related functions.
- The framework developed here supports the reading and translation of conceptual designs, object-oriented data modelling, query modelling implemented with both declarative and procedural constructs, and data type checking and inference required by various models. The framework is an effective tool for the database designer as it integrates and aids many activities.

1.2 Plan of the Thesis

The second chapter surveys the conceptual EERM language. Any implementation details are avoided; for example *n*-ary relationships are shown as such and not decomposed.

The third and fourth chapters cover the object-oriented paradigm by going through its many themes and variations to synthesise a data centric data model. In the fifth chapter a definition of an object-oriented database is given. It also includes an overview of ODMG ODL and an ODMG compliant OODBMS, called EyeDB. The latter is used for running examples. Our own critical overview of ODMG ODL is given at the end of the chapter.

Chapter six surveys the very useful developments in Datalog and a logic with object-oriented semantics, called F-logic. Flora-2, a logic-programming language implementation of F-logic, is described here too. Chapter seven surveys integrity constraints, an important component of data and query models, and gives their semantics is given in first order logic.

The bulk of our research effort is presented from chapter eight to twelve. Chapter eight has a detailed description of a framework built purposely to enable and integrate the many parts that comprise this research body. Also the chapter gives encodings of database artefacts; for example classes, and *ISA* relationships. Many of the database features require implementation and enforcement in our framework; these are programmed in Flora-2.

In chapter nine we indicate how an EERM conceptual diagram is read and encoded into our framework; also a working procedure is presented that converts the conceptual encodings into a set of ODMG ODL constructs, specifically EyeDB constructs, after necessary transformations of the EERM diagram are done.

In chapter ten one finds how data type checking and type inference are built into our framework. This module is necessary because Flora-2 does not enforce the data type signatures specified in programs written for it.

In chapters eleven and twelve we develop our procedural algebra; the first chapter has operations that have a relational origin while in the second chapter the operators are based on the features related to object values and functional methods. For each operator one finds examples, its data-type signatures of its input and output, the pre-conditions and post-conditions of the operator, and the actual logic-programming based implementation. The algebraic properties are also given.

In chapter thirteen a number of issues relating to query processing and optimisation are described. Special emphasis is made on techniques that make best use of the artefacts in the underlying data and query model. Furthermore a conversion of declarative OQL queries into an algebraic expression using our operators is given. The procedure is also amended with basic, but effective, improvements.

Chapter fourteen presents a summary and conclusions of our work. Indications of current status and future work possibilities are given too.

Chapter 2

Conceptual Modelling

Chapter 2 – Conceptual Modelling

Stonebraker states "In contrast to data normalisation, the Entity Relationship Modelling became very popular as a database design tool" in [STONE98]. The Entity Relationship Modelling is labelled as conceptual because it is devoid of any physical database artefacts (e.g. data indexing methods). It is also evident that a good number of structures and constraints are understood and caught very early in the conceptual design. Entity Relationship Model modelling is also adopted in many CASE tools as a basis for generating a relational database after data requirements are validated and verified.

2.1 – ERM and EERM

The *Entity Relationship Model (ERM)* introduced by Chen [CHENP76] and formalised by Ng [NGPAU81] is a high-level conceptual data modelling language. Since its introduction, a number of additions to the original graphical language have occurred, with a case in point (and of interest here) being the Enhanced ERM (EERM). Ramifications from the ERM are numerous, with a significant example being the class diagram found in the Unified Modelling Language (UML) [BOOCH99].

Other extensions and revisions followed; some are very specific data models as in spatial and temporal extensions; yet others are closer to a database model rather than remaining abstract as in Chen's presentation.

The ERM, in the context of application-development analysis and design, focuses on logical units and their constraints. Also in many structured methodologies for application development the ERM is used in supplementary and complementary fashion with functional descriptions and data flow paths. Once an application analysis and design is stable then the first step of development is channelled toward the realisation of the conceptual design by mapping onto DBMS and programming language specifics. The ERM language is data model independent and consequently it does not particularly favour the relational over the object-oriented model and therefore specific procedures to map the high-level conceptual design onto the constructs of any particular data model are needed.

The main constructs of the ERM are the entities and their relationships. The contextual use of an ERM assumes that each entity and relationship has many instances. It is

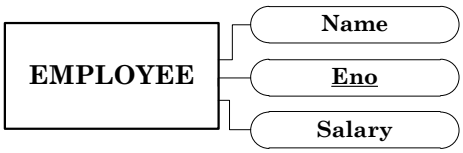
important to note that each construct is adorned with properties and constraints. Later additions to the "basic" ERM introduced other data modelling constructs (e.g. specialisation and aggregation) and their associated constraint's descriptions.

Most of the artefacts that make up an ERM are depicted graphically, for example rectangles, diamonds and ovals, and the edges between these nodes represent associations between them.

2.1.1 Entities

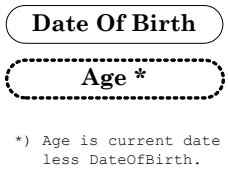
An *entity* in an ERM graph is a named construct representing a real or abstract concept in the application domain of discourse. Historically the area of Artificial Intelligence gave the origin of this denotation. For example an **EMPLOYEE** is an entity in a project description information system. Each entity name in a graph is unique. An entity is presented as a named rectangle in the model's graph. Each entity represented in the model is expected to have many instances but these instances are not represented in the ERM.

An entity is also described by a number of properties called *attributes*. Each attribute has a name, and although not depicted in a graph (or colloquially called a diagram) it is annotated with a value domain. For example in the case of the **EMPLOYEE** entity likely attributes would be **NAME**, **ENO** and **SALARY** with value domains of character string, integer, and real number respectively. Each attribute is depicted as an oval (or a rectangle with softened edges) attached to its entity. Property names have to be unique within an entity. Entity instances would then have to be assigned values to these attributes from values of their associated domain. The model does not limit on which set of basic domains are used.



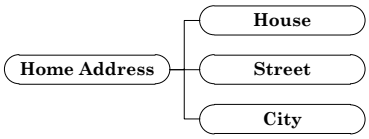
Attributes come in a number of flavours and with specified constraints. For example an attribute (or set of) whose value(s) uniquely distinguish any instance from all others of the same entity is called the *primary key (set)*. To represent this constraint the respective attributes names are underlined. Each entity can have one primary key set. In the case of **EMPLOYEE** the primary key is **ENO**.

Some attributes directly derive their value from other attribute values present in the same instance. For example, given an instance's **DATE_OF_BIRTH** value then it is easy to compute its current **AGE** (i.e. without the need to store and keep up to date the relative attributes value). These attributes are the *computed attributes* and require the designer to draw the oval with a dotted line. Also it is sensible to include a succinct note on how this attribute value is derived.



Another aspect of attributes is whether an attribute is considered to be single valued or multi-valued. In most cases one expects a *single value* from the domain; as in **SALARY** and **DATE_OF_BIRTH**. In some other cases an attribute takes many values from the associated domain. In this case we have a *multi-valued attribute* and these are represented with double ovals. An example would be a property that describes the **SKILLS** of an employee in which we envisage that an employee instance can have zero or many skills.

Sometimes a sub-set of an entity's properties tend to have a high affinity and it is advantageous to compose these together into what is called a *composite attribute*. For example assume we have the following properties, **HOUSE**, **STREET** and **CITY**. These three can form the composite attribute **HOME_ADDRESS**. The notation used is of a composite attribute attached to its entity and the parts attached to the composite attribute.



Each attribute property could have a number of varieties. For example it is conceivable to have a computed attribute that is multi-valued and composite.

2.1.2 Relationships

The other major part of an ERM is depicting relationships between entities. A *relationship* is a named association between a number of entities that is governed by rules and sometimes qualified with its own attributes. A *relationship instance* is a particular case of a relationship and is qualified by the entity instances that it relates. For example given the entities **TEAM** and **EMPLOYEE** are related by an **EMPLOY** relationship and #1056 and #118 are instances of the **TEAM** and **EMPLOYEE** entities, and then #1056 **EMPLOY** #118 is an **EMPLOY** relationship instance.

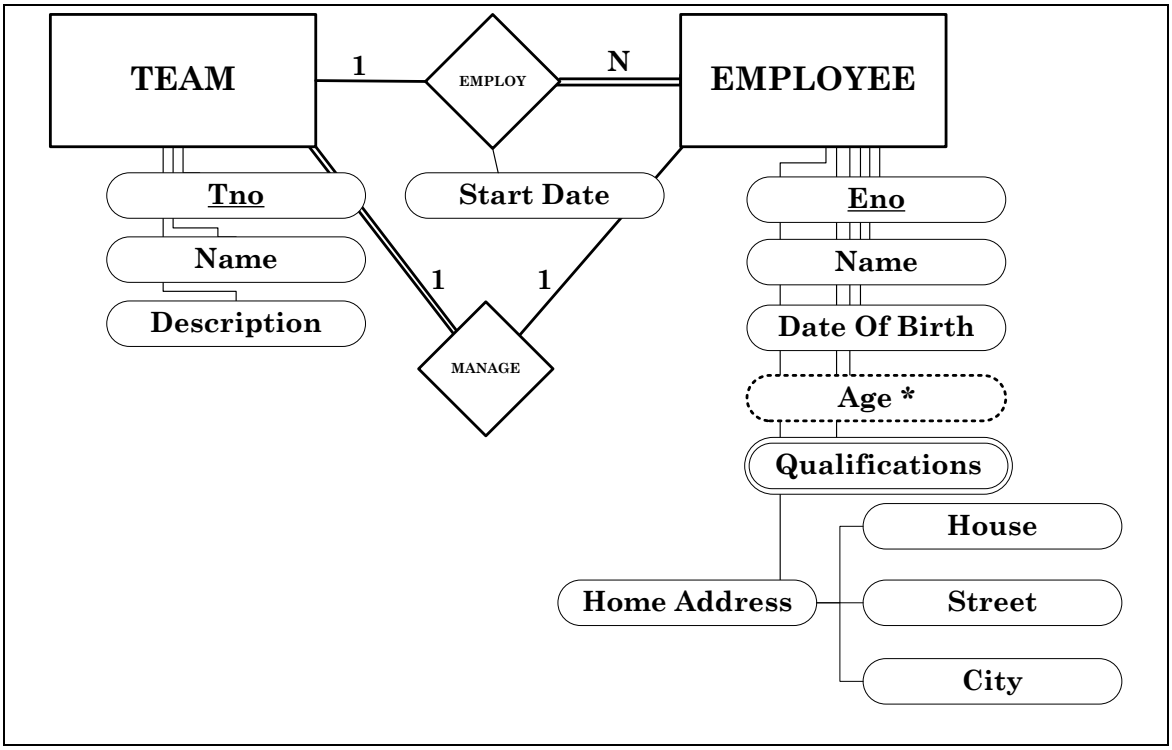


Fig. 2.1 – ERM with entities and relationships

As stated in the definition of a relationship its instances must adhere to its rules. There are three types of rules that relate to a relationship definition and these supplement the relationship depiction in an ERM (a diamond with an edge connecting each associated entity – see figure 2.1).

1) The first constraint deals with *enumerating* the entities of a relationship. Consequently any relationship instance composition must have each respective entity instance coming from the respective entities. The *degree* of a relationship is the number of entities that define the relationship. If there are two then the relationship is called *binary* and if there are three then it is called *ternary*. In the general case of *n*

entities the relationship is called an *n*-ary. If an entity is repeated in a relationship definition then the relationship has an additional adjective – *recursive*. The most popular type is the binary relationship. The **EMPLOY** is a binary relationship and its participating entities are **TEAM** and **EMPLOYEE**.

2) The second constraint specifies, given a relationship between entities, the count of an entity's instances that can participate in a relationship instance. The most common cardinalities are the 'one' and the 'many' times. For a binary relationship the possible number of these types are four; namely the *one-to-one* (1-1), *one-to-many* (1-N), *many-to-one* (N-1), and *many-to-many* (N-M). For example the **EMPLOY** relationship in figure 2.1 is a one-to-many as firstly an instance of team can be associated with many instances of employee and secondly as an instance of employee is only associated with at most one team. In the modelling diagram the cardinalities are marked by including the symbols "1" and "N" on the respective edges of the relationship. In the same figure the **MANAGE** relationship is a one-to-one.

3) Another important constraint relates to the level of participation of an entity's instances in a relationship. The options are *total* and *partial*. Total participation implies that all instances must participate in at least one relationship instance. For example in the **EMPLOY** relationship in figure 2.1 it is expected that all **EMPLOYEE**s are employed with a team but it is not necessary for all teams to have an employee on their books. A partial participation implies that an entity instance may not be present in any relationship instance. Consequently, for any binary relationship, there are four possible cardinality participation constraints; namely total-total, total-partial, partial-total and partial-partial. For a total constraint the edge leading from a relationship to an entity is drawn as a double line. When writing relationships it is sometimes convenient to write 1(p)-N(t) for a 1-N relationship with a partial and total participation as in the **EMPLOY** relationship. In the case of the **MANAGE** relationship between **TEAM** and **EMPLOYEE** one writes 1(t)-1(p).

It was earlier indicated that a relationship might have attributes that describe it and consequently each relationship attribute has a domain and other qualifiers mentioned before for entity attributes. This is easily denoted in an ERM diagram by connecting an

attribute to the respective relationship graphic (i.e. a diamond). A typical example of a relationship attribute is the case of the **STARTING_DATE** for the **EMPLOY** relationship (see figure 2.1). The context of attributes attached to a relationship rather than an entity is obvious but it is in fact an area that shows a weakness of the *modus operandi* of modelling through ERMs. For example in a binary 1-N, the relationship attribute could be moved to the many cardinality entity (i.e. **EMPLOYEE**). This is of course a "legal" arrangement but of questionable style even if the ERM building rules are not compromised. (This and other problems with ERMs are discussed in a later section in this chapter).

2.1.3 Other ERM Features

What we have seemed up to now is a basic description of ERMs. There are still some other points to explain. Two of these are weak entities and ERM notes.

2.1.3.1 Weak Entities and Weak Relationships

It is not always the case that entity's instances describing some application have an independent existence from all other entities. For example if for the **EMPLOYEE** entity we like to hold details of an employee's **WORK ASSIGNMENT**s then purging an **EMPLOYEE** instance presumes we are purging all **WORK ASSIGNMENT** instances related to the same **EMPLOYEE** too (see figure 2.2). Clearly there is an asymmetric dependence between **WORK ASSIGNMENT** instances and the respective **EMPLOYEE** instance. To address this modelling requirement the ERM introduces another structure called a *weak entity* whose instance's existence is related to a "normal" entity's instance existence. In this case **WORK ASSIGNMENT** is a weak entity dependent on **EMPLOYEE** entity.

The relationship between a weak entity and its normal entity is also specialised – it is called a *weak relationship*. The weak relationship is called **UNDERTAKE** in this example. At this point the entity and relationship constraints (on the weak relationship and the participating entities) need consideration. Firstly it is obvious that the weak entity's relationship to its associated normal entity must be total. Also the primary constraint of the weak entities instances scope is not necessary for all the instances of the weak entity but rather for all the weak instances related to a normal entity instance. In some literature this case is known as a *partial key*.

For depicting a weak entity and weak relationships we draw the rectangle and diamond with a double line. If the weak entity has a partial key (rather than a normal primary key) then the underlining is drawn dotted rather than continuous. (See figure 2.2).

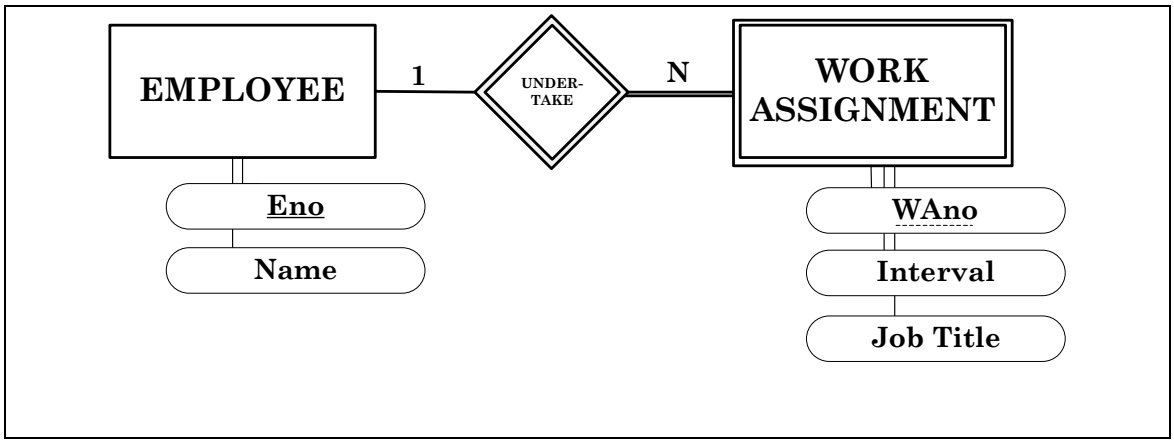


Fig. 2.2 – ERM with entities, weak entities and weak relationships

The cardinalities acceptable in a weak relationship are only one-to-many and one-to-one. The relationship must be a total participation on the weak entity. If a many-to-many weak relationship is required then one is expected to consider other modelling constructs; for example an aggregation relationship (more in a later section). Likewise if a weak entity seems to require a relationship with any other entity then again it is best to consider aggregation.

The introduction of a weak entity and a weak relationship is not only a new type of structural representation. In fact it is the first example in the ERM where we have a transitional action associated with a state change (i.e. a change of the data values).

2.1.3.2. ERM "Notes" (or Annotations)

We already have seen the utility of having notes in a diagram (e.g. to explain the computation of a derived attribute). There are also times when notes are required and in fact become a necessity. For example if some business rule requires a structural configuration constraint that isn't caught by our ERM modelling then there is no choice but to write explicit notes for the designers to understand. Unfortunately this may entail translating these notes into programming constructs that are not known to a database schema.

2.2 Enhanced ERM (EERM)

Since its inception the ERM has had a number of additions. Some of these are motivated by the desire of analysts to capture and analyse better the modelling aspects of an application. This demand is more relevant in areas where data modelling is more contrived – for example a **CLIENT** entity could have a number of attributes that are applicable only to a subset of its instances. These scenarios, although implied to be more sophisticated than plain business applications, became more numerous as information systems supports ever more varied and wider sets of requirements. Nonetheless the original ERM context of having each artefact representing many instances still prevails.

A string of additions over the basic ERM are those coming from the semantic data modelling [SMITH77 & SMITH77B] and are collectively called *Enhanced ERM* (or *Extended ERM* is sometimes cited too). The additions that are of interest here are specialisation and generalisation, aggregation and variant records. Unfortunately there isn't much agreement on the meaning, usage and depiction of these and consequently we will have to cite the source that the construct is coming from without implying that it is the only (or best) representation.

2.2.1 Entities and Sub-Classes

If in an application domain an entity seems to exhibit distinct clustering of its instances then it is best to partition the instances into subsets. The basic idea is attributed to Smith and Smith [SMITH77]. A *subset* has all the properties defined for the class that it is sub-setting and possibly defines more of its own; this is often called *sub-classing*. Each instance in a subset is also an instance in the original entity set of instances. For example the **EMPLOYEE** instances can be divided into either technical employees or administrative employees or neither (see figure 2.3). This is an example of sub-classing where **TECHNICAL** and **ADMINISTRATIVE** (**ADMIN** for short) are sub-classes of **EMPLOYEE;** and **EMPLOYEE** is loosely called a parent class. Both sub-classes would inherit attributes and relationships from their parent entity and are allowed to independently add their own attributes and relationships. Consequently sub-classing allows us to specify relationships concisely; for example a **TECHNICAL** instance (not just any **EMPLOYEE**) can use a piece of **HARDWARE** or **SOFTWARE**. Also sub-setting allows the entities to be organised in an *entity's hierarchy*.

There are a number of options to this sub-setting. Firstly is sub-classing *partial* or *total* (see figure 2.4)? Secondly is sub-classing allowed to *overlap* or not (i.e. *disjoint*) between sub-classes of an entity (see figure 2.5)? If, for example, we have an **EMPLOYEE** who can be a **TECHNICAL** or an **ADMIN** instance, and there might be **EMPLOYEE**s who are neither, then our sub-class relation has a partial participation and also has disjoint sub-setting. Thirdly an instance can be included in a sub-class either by way of its attribute(s) value called a *predicate defined* instance on its subclasses or by making it an arbitrary instance of a subset (where this is called an explicit *semantic relationship*). An example of predicate defined sub-classing follows: let us assume we have two sub-classes, **PERMANENT** or **CONSULTANT**, of **EMPLOYEE** which has an attribute called **STATUS** that holds an indicator of the pay structure. In this case each **PERMANENT** and **CONSULTANT** instance should have the following value for the **STATUS** attribute of "full-time" and "part-time" respectively.

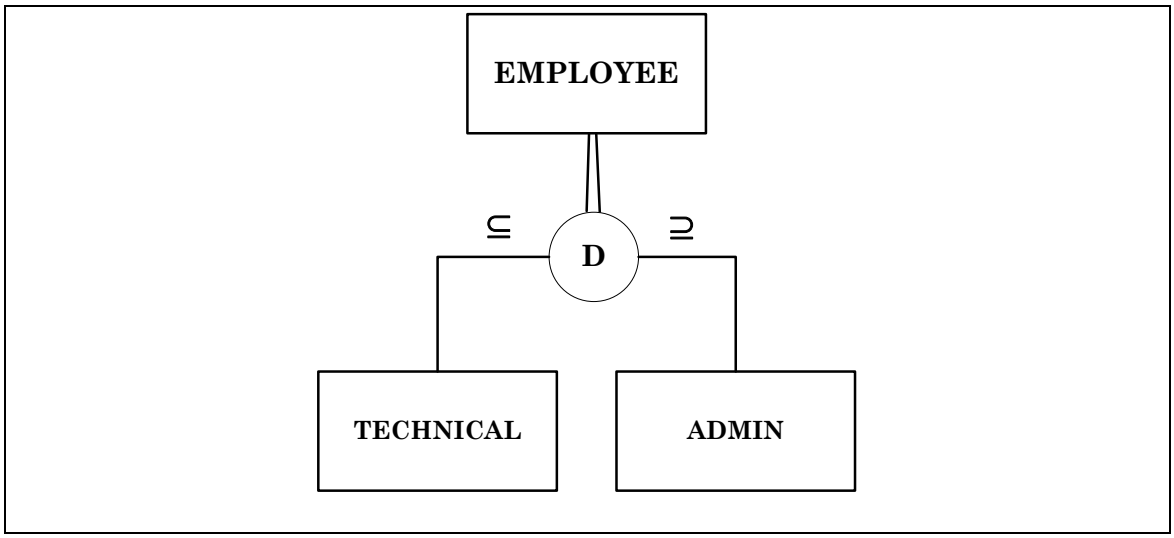


Fig 2.3 – EERM with Employee and its subclasses related through a disjoint and total relationship.

To draw the sub-class relation between a class and its sub-class use a circle and pair of edges to connect the relevant entities. Within the circle we print a "D" for disjointness and an "O" for overlapping – see figures 2.3 and 2.5. Also on the edge closer to the sub-class we draw the sub-set symbol (i.e. ⊆) with the pointed edges in the direction of the sub-class. If the relation is total then the edge from the circle to the class is drawn in double line. If a sub-class relation is predicate defined then that predicate is written on the main edge (the edge from the class to the circle) and the value that satisfies on the edge from the circle to

the sub-class. As for drawing styles there are two main references: Elmasri [ELMAS10] and Teorey [TEARE05].

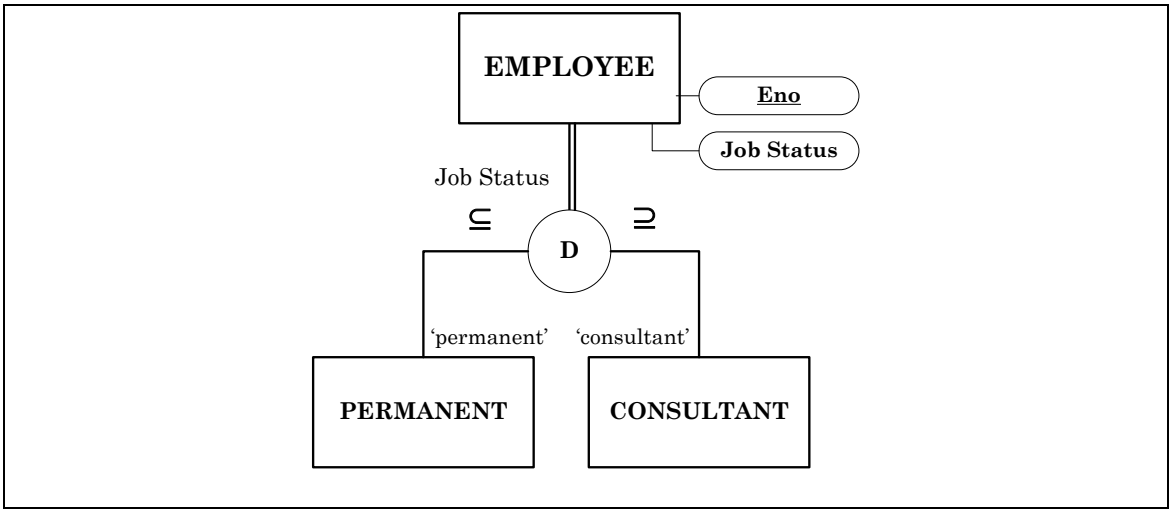


Fig 2.4 – EERM with Employee and its subclasses related through a disjoint, total and predicate defined (i.e. Job Status) relationship.

In some references known for presenting sub-classing in EERM (e.g. [HULLR87]) it is allowed having a number of sub-setting "relationships" radiating from the same entity. In figure 2.5 we have done this; **EMPLOYEE** has a theme of subsets based on **TECHNICAL** and **ADMIN** and another theme based on the subsets by **PERMANENT** and **CONSULTANT**. These two themes are independent from each other. This multiplicity of sub-setting does complicate the diagram and the entity's hierarchy.

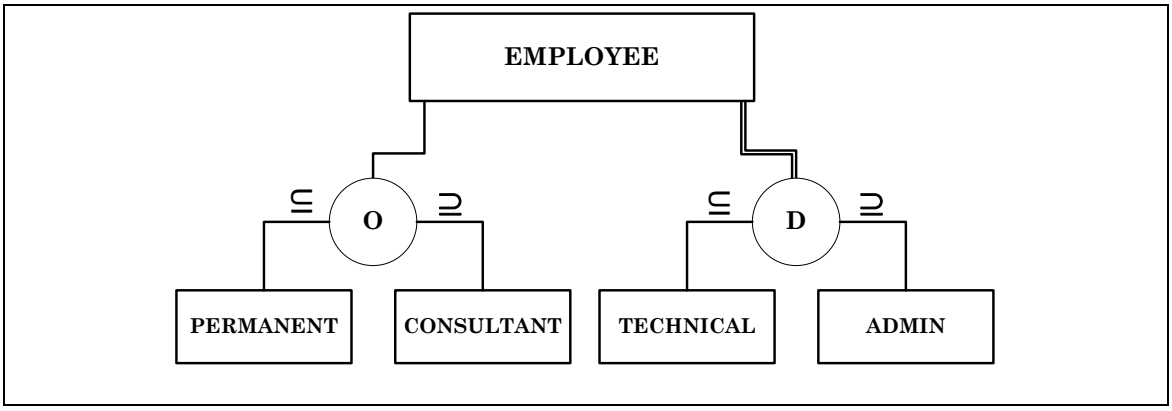


Fig 2.5 – EERM with multiple sub-setting relationships

There is of course the ever present problem associated with inheritance – *multiple inheritance*. This is possible in EERM and the resolution of conflicts is usually based on name overriding in an arbitrary manner. Most references on EERM state that the analyst

is free to restrict his diagram to single inheritance; albeit making the design cluttered and repetitive.

As for entities and relationships, the sub-classing construct comes with a series of constraints that all instances involved must adhere to. Some of these constraints are static, while some others are transitional; in references [ELMAS10] and [TEORE05] one finds the following. The static constraints include:

1) There should be no cycles in the entity-hierarchy through the sub-classing relation.
2) If sub-classing is restricted to a single inheritance then each entity can only be a sub-class of at most one entity.
3) If a sub-classing relation is total then all instances of the parent entity must be an instance of at least one sub-class.
4) If the sub-classing relation is predicate defined then the discriminating value for each sub-class must be non-ambiguous. Also each instance of the sub-class has its defining attribute value satisfies any explicitly stated value in the diagram.
5) If the sub-classing relation is arbitrary (not defined with a predicate) and disjointness is specified then each instance of a sub-class in an instance of at most one sub-class.

The transitional constraints on the sub-classing relation include:

1) On introducing a new instance to an entity that has a total sub-class relation then by implication the same instance has to be made an instance of one of the sub-classes. Additionally on introducing a new instance to an entity that has a predicate defined and total sub-classes constraint then by implication that instance is also made an instance of the matching sub-classes.
2) On purging an entity instance then all the associations in its sub-classes are also deleted.
3) On changing the values of an instance's attributes that affects the value that determine a predicate defined sub-class relation then that instance becomes the instance of the current sub-class that matches its new value. Migrating instances from one sub-class to another requires *ad hoc* procedures because other attribute values might not satisfy destination class properties.

2.2.2. Categories / Union Records

A useful addition to the ERM is the category construct (as introduced by Elmasri *et al* [ELMAS85] and also found in NIAM models as an exclusion constraint [VERHE82]). Let us use our **EMPLOYEE** entity with **TECHNICAL**, **ADMIN** and **MANAGER** sub-classes. If we want to represent a relationship between an employee and a piece of work then we can relate **EMPLOYEE** and an entity called **WORK**. If it transpires that only instances from **TECH** and **ADMIN** should relate to **WORK** instances then we have to reconsider the diagram. We could leave the relationship between **EMPLOYEE** and **WORK** and write a note that prohibits the participation of **MANAGER** instances. Or we can create a new entity, if multiple inheritance is allowed, that sub-classes **TECH** and **ADMIN**, say called **TECH_ADMIN_EMP**, and relate this to **WORK**. Unfortunately this is not semantically correct as the instances of **TECH_ADMIN_EMP** include all attributes of both **TECH** and **ADMIN**. Also if the two entities are not really relate-able through a sub-class relationship, then creating an entity that inherits from both is generally not in good style.

A better solution is to use the *category* artefact whereby a "new" entity denoted in the diagram represents the *union of the instances* of a number of entities and this union is a pool of instances that can participate in a relationship. The meaning of this construct is slightly different from the previous relationships as entities participating in the union would generally have different primary key set constraints; one solution is to include details of both entities and instances in the relationship instance details. The notation to represent the category concept uses the circle with a letter "U" inside it – see figure 2.6. Also the category construct can be qualified as total or partial. We are not too comfortable about introducing a "new" rectangle (as in [ELMAS85]) and really prefer the NIAM exclusive notation (i.e. connect the diamond to the circle which is connected to each entity participating in the union and connect the emanating relationships with an arc). There are two reasons for this: firstly introducing an entity which is really a "view" of other entities and secondly some flexibility is lost as there can be only one cardinality ratio for all entities participating in the union.

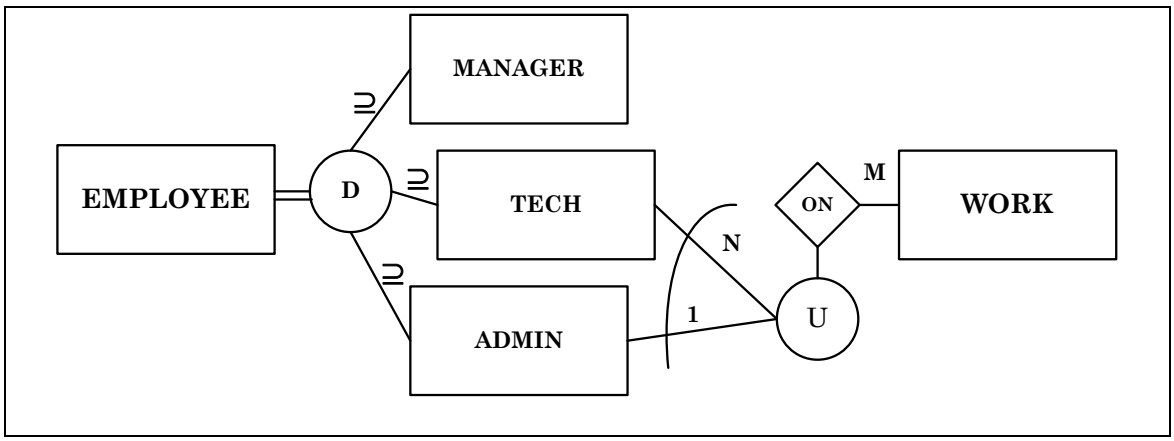


Fig. 2.6 – An EERM with a category (or EOR) artefact – i.e. arc.

2.2.3 Aggregation

Aggregation builds an instance out of a number of other instances through a sophisticated relationship between the "whole" and its "parts" instances. This relationship is usually transitive but not necessarily. Aggregation, sometimes colloquially called the *part-of* relationship, is at a higher level than entity composition. Although this rich construct has been known for a long time (through Smith and Smith's work in [SMITH77b]) few diagrams actually use it. Later and significant literature includes Steel's work on Lisp [STEEL84], the MCC Orion initiative [KIMWO89G], UML [BOOCH00], and Barbier's [BARBI03]. A telling rendition of aggregation through a *'bill of material'* diagram for a bicycle is shown in figure 2.7. While the front wheel brake unit is a part of the whole bicycle, the same unit is in itself a whole, at least in terms of spare part sales, as it is made up of other sub parts.

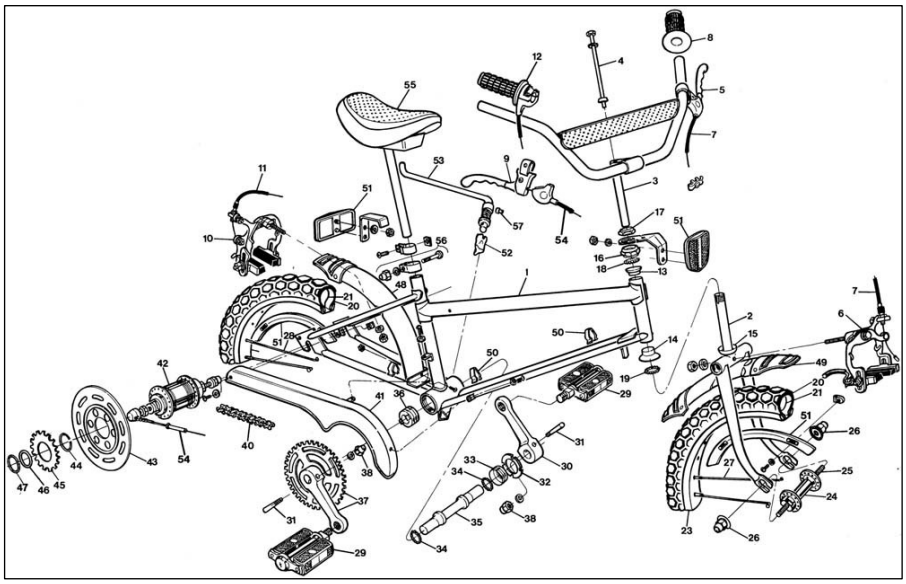


Fig. 2.7 – Exploded drawing of a bicycle (Source: Raleigh Grifter DL60 Exploded Drawing from 1977 Raleigh Dealer Manual).

We need to have an aggregated relationship which captures: 1) sharing and independence of "parts" by the "whole"; 2) total or partial participation of "part" instances; 3) attributes to each manifestation of the "part" in a transitive relationship; 4) attributes of the whole transitive closure. In an aggregated relationship sharing and independence are orthogonal and consequently we have four possibilities – see figure 2.8.

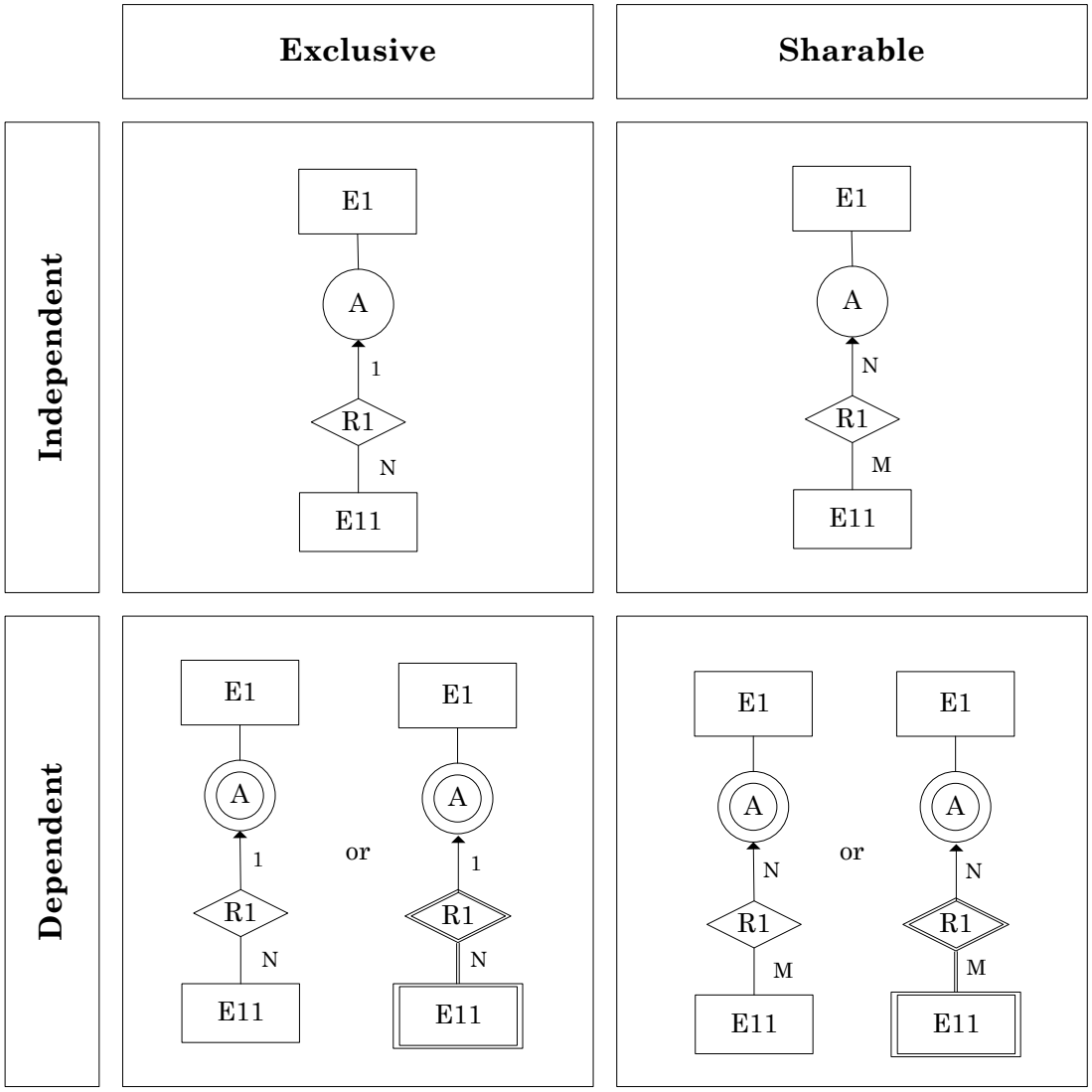


Fig. 2.8 – Aggregation in EERM (no transitivity is shown)

As for notation there are two main models, one found in Elmasri [ELMAS10] and the other in Teorey [TEARE05] which is the one adopted here and it uses a special circle with an "A". In figure 2.8 we give the four combinations of an aggregate relationship through the latter notation and using sharing and independence to qualify relationship. It is important to note that sharing is connected by a many-to-many relationship. What do the four patterns

yield in terms of the aggregation relationship? The following table gives actual examples; see table 2.1.

Independence	**Sharing**	**Example**
Independent	Sharable	Cyclocomputer applet that tracks effort in terms of distance and climb.
Independent	Exclusive	A bicycle tyre.
Dependent	Sharable	Frame spray paint.
Dependent	Exclusive	Lock nuts (following advice that lock nuts are to be used only once!)

Table 2.1 – Four aggregation patterns through independence and sharing.

An example diagram that can meet some of the data modelling requirements of an aggregation in the manufacturing of books is in figure 2.9. In this case there is no part-of transitive relationship. Another example relationship that describes a typical bill of material composition is given in figure 2.10; it has a transitive part-of relationships.

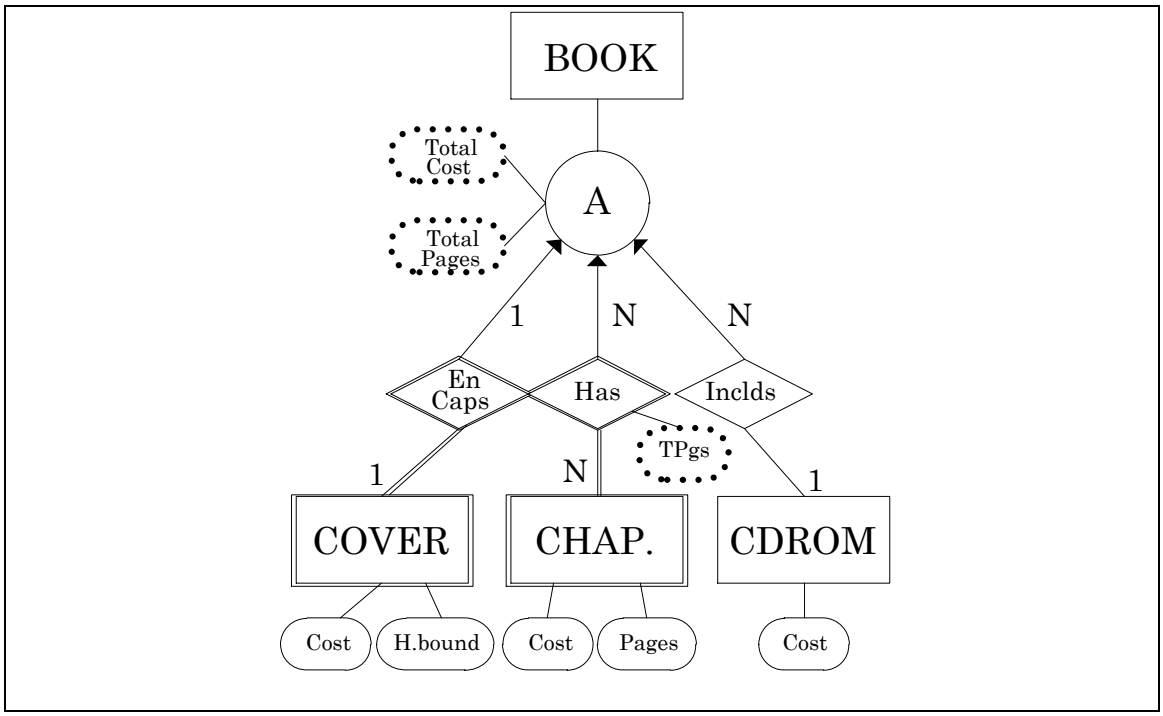


Fig. 2.9 – A book construction EERM with a simple aggregation relationship.

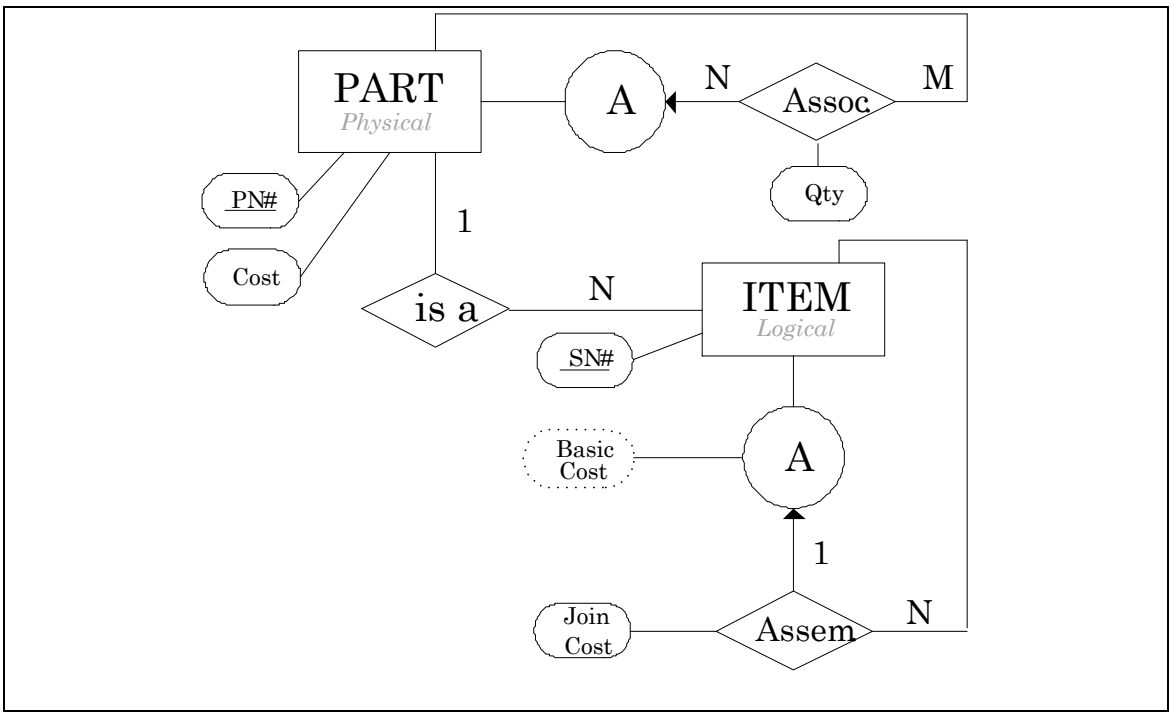


Fig. 2.10 – A bill of material EERM with transitive relationships. Note that entity item represents tangible parts, while parts entity is a catalogue description of items.

For an aggregation relationship to become distinctive from a normal relationship one has to consider two aspects. The first has to do with instances of a relationship; specifically each instance of a normal relationship is independent from any other instance but in the case of an aggregation relationship many instances are related through transitive closure. The second aspect is the asymmetric role of an aggregation relationship that contrasts with the symmetrical nature of a normal relationship. Consequently before introducing an aggregated relationship the analyst must carefully consider these two aspects. In practice these distinctions might not offer conclusive arguments in favour of inclusion of an aggregation relationship in a diagram.

There are a number of implicit constraints associated with an aggregation relationship representation in the EERM diagrams. Some examples follow (refer to figure 2.11).

a) If a "whole" instance is deleted and that instance is in a transitive aggregation relationship that has an encoded non-shareable and dependent part-of relation with other instances then all these "parts" are purged too. Consequently a portion of the graph, the sub-graph rooted at the deleted entity instance of the graph, is deleted. In figure 2.11 deleting instance 'c' reduces graph A) to C).

b) Assume we have an aggregation relationship between the "whole" and its parts in a transitive aggregation relationship that has an encoded independent part-of relation; then deleting an instance would generate a number of transitive closures. That is, the original graph can break into "smaller graphs". In the example deleting instance 'c' reduces graph A) to D).

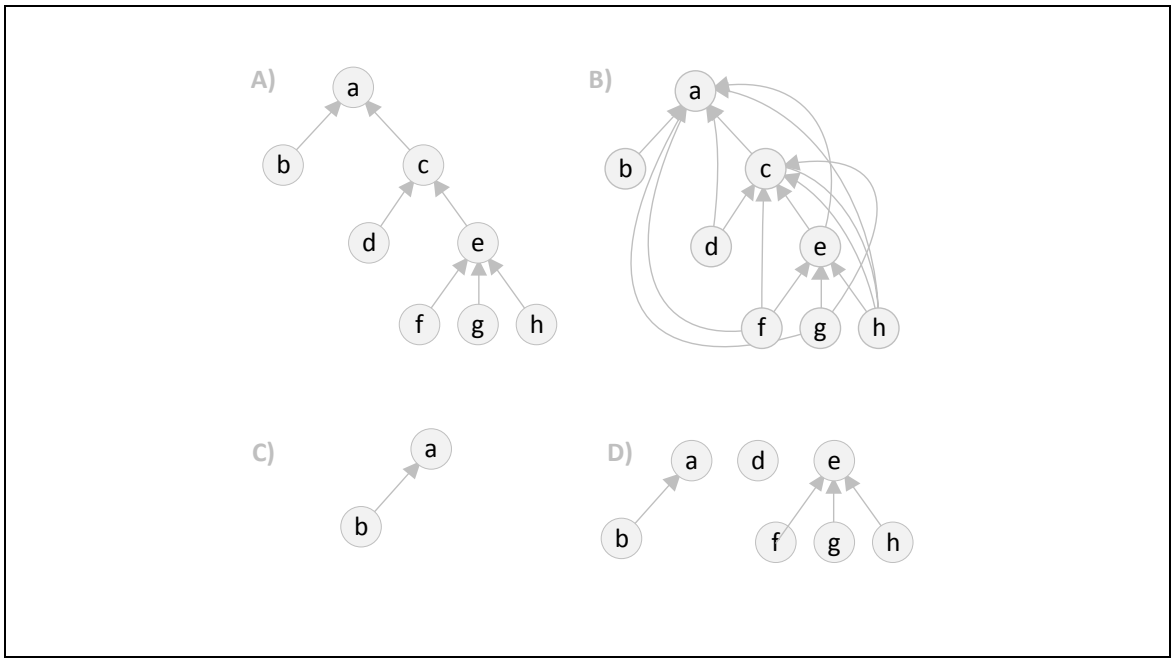


Fig. 2.11 – A number of instances (named 'a' to 'h') are related with a part of relationship – shown as edges A). The instances transitive closure is represented in B). Graphs C) and D) are the result of deleting instance 'c' with different part-of constraints.

Some "convoluted" constraint requirements are usually identified during design; for example the costs of "parts" should not exceed 30; the depth of transitive closure tree should be between 3 and 5, the total distance of a travel plan should not exceed 200 units. Most of these have to be included as notes to the EERM; notice the use of an entity's computed attributes in figure 2.9 and 2.10 are assigned values computed by tallying the sub-graph.

Some notes of caution have to be mentioned here. Firstly, it is possible to create legal diagrams that do not make intuitive sense. Secondly, some combinations of the above features can be represented as a simpler ERM (e.g. a N(p)-1(t) between a weak entity and its related entity can be represented much more simply as a basic aggregation relationship). As for the former we assume a pre-processing of an EERM can detect these

"aberrant" relationships, while for the latter we also assume that the simpler solution is drawn.

Nonetheless the biggest problem is that these advanced constructs require more than malleable structural rules; in fact good transitional data description constructs are required.

2.3 How and How not to Use ERM

ERM is a high-level language and we have seen how its diagrams capture a reasonable level of structures, relationships and constraints. From a software engineering perspective there is yet another important aspect for using ERMs for database analysis and design: how to go about creating the diagrams – i.e. a *methodology*. If the use of ERM is part of a larger methodology (e.g. SSADM) then it is best to follow the guidelines of the methodology and representation. If, on the other hand, we are interested in building the database and depicting its role in the overall functionality, then a methodology as presented in Batini's *et al* textbook [BATIN92] is a well-known technique.

In general an ERM is used in top-down design methodologies where these are given an informal description of the data requirements and the diagram is built by refining and building on successive versions of the diagram.

The general advantages of using ERM's proposals in database design are the following.

1) ERMs are high-level languages that are relatively easy to read by a wide spectrum of users (analysts, designers and selected end-users).
2) ERMs are independent from any particular data model and especially any database physical constructs (e.g. indices, or data placement policies).
3) ERMs describe a high proportion of the data structures, relationships and static constraints that an information system requires. These models are in fact used in describing the conceptual and external schemas of the ANSI/SPARC three-level data architecture. Also some papers use ERMs as a basis for building co-operative information systems (e.g. a loosely couple multi-database framework); a a seminal paper is [HEIMB85].

4) ERMs are also used as a test bed during the analysis and early design phases. The diagrams are useful to verify and validate an information system's high-level requirements.

5) ERM's constructs are not fudged or dictated by the target database model; for example, with a relational data model an M-N relationship is broken into two 1-N relationship and a resolving entity.

The main disadvantages of using an ERM approach in database design are the following four.

1) A properly drawn ERM does not guarantee it is devoid of data redundancy. In fact it is highly appropriate to re-design critical areas of the diagram through bottom-up data analysis (especially if the target data model is the relational then at least Boyce Codd Normal Form should be established). Also some patterns in a diagram require the analysts to re-check these (more specific examples, such as connection traps, are given in the following sub-section).

2) For information systems that have a "large" design the size of the diagram and interconnections make it hard to read and maintain (e.g. re-draw). Some researchers (a case in point being Elmasri [ELMAS10]) do attempt to "package" the diagram through a partitioning into simpler diagrams. Also some highly data-driven and modular systems (with an end-less list of start-up and operational parameters – e.g. SAP® accounting package { WWW.SAP.COM }) makes the diagram a "complex" graph. Consequently EERMs do not scale with size of design.

3) There is an issue of drawing standardisation. This is made worst by having Graphical User Interface design tools that are too tightly coupled to a specific DBMS.

4) A query model, based on ERM, is not available *per se*. Nonetheless, subsets of ERM diagrams are used to describe the structure of an external schema – a view. An interesting development in UML from OMG is Object Constraint Language (OCL) [OMGRP06] and Query/View/Transformation (QVT) [OMGRP11].

2.3.1. ERMs and CASE tools

A number of tools are available to the designer who wants to adopt ERM modelling. These provide drawing, and conversion to a database model – mostly relational and a few object

relational. Popular examples include CA Erwin { WWW.ERWIN.COM }, Toad Data Modeller {WWW.QUEST.COM/TOAD-DATA-MODELER}, and DB Designer (which is open source and recently renamed) { WWW.FABFORCE.NET/DBDESIGNER4 } – see figure 2.12. The better ones allow for customisation but mostly 'improve' on Chen's original diagram artefacts. A useful feature is the ability of a tool to read a database data dictionary and reverse engineer an ERM.

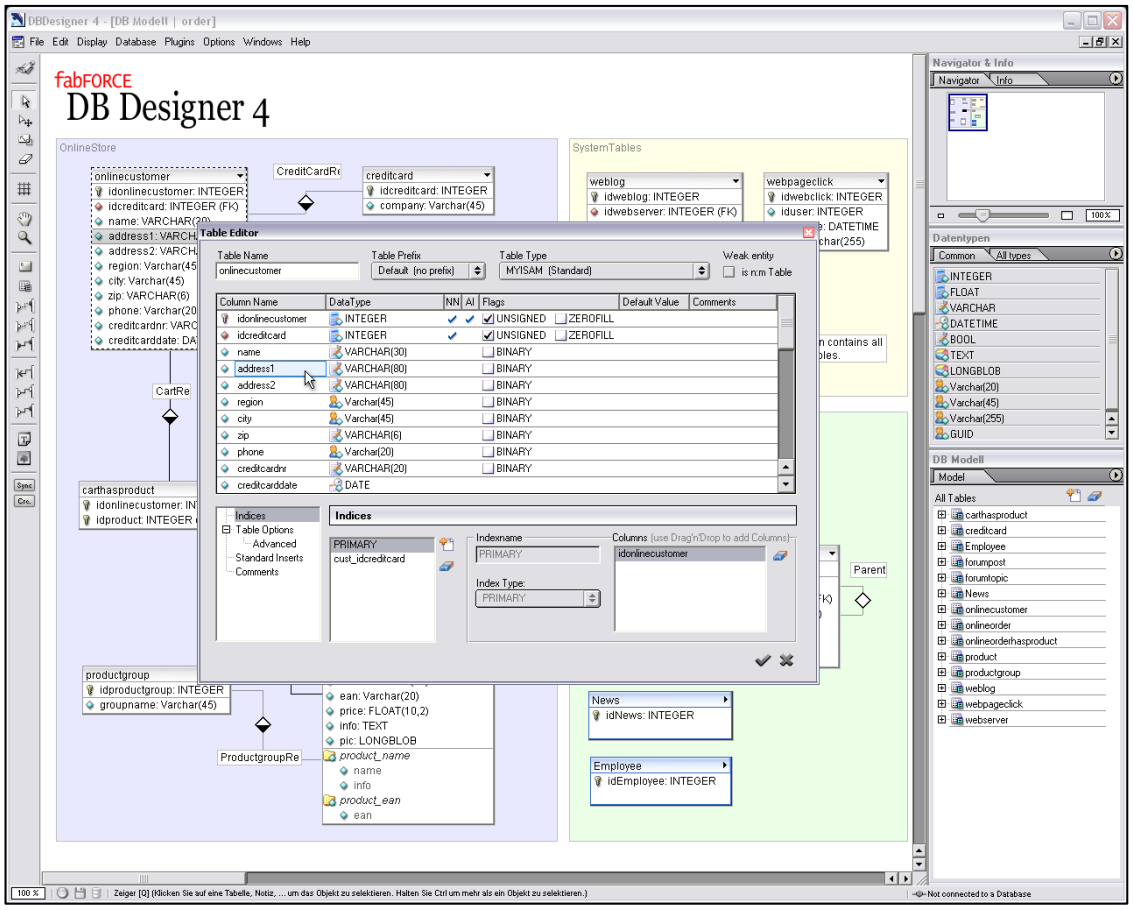


Fig. 2.12 – Screen dump from one of the conceptual design CASE tools.

2.4 Problems with ERM/EERM diagram

A perennial problem with ERM is whether a fact in the information system is presented as an entity or as an attribute in another entity. A case in point is an address abstraction. The design heuristic taught for resolving this issue is checking in the requirements whether an address ever needs to be decomposed into further parts. If it is then convert the attribute into related entity. Another aspect to consider when making this decision is the data quality level requirement; for example does the address postcode need to spatially coincide with that of the Post Office. For example, if our information system does not

match on addresses but feeds into another system which does that (e.g. a data warehouse [INMON02]) then the address is better represented as an entity.

Another nagging problem is the admission of *"null"* values. In databases a null occurs mainly in two situations. The first it comes about when an instance does not have an applicable value for an attribute (e.g. a company does not have a birthday). The second comes about from the lack of knowledge about an entity's attribute (e.g. a person has a date of birth which is not known in the system). The occurrence fudges the meaning of instances, as the presence of a null in an attribute's value is ambiguous. It is best to "design out" the possibility of null assignment in parts of the ERM where computations and comparisons are intense.

There is another level of problems with ER diagrams. These are patterns that, based on arrangements of entities and relationships, although structurally correct, do not completely convey the meaning of the domain of discourse – these patterns fall under the generic term of connection traps. Also some constraints are either too specific or too general; and consequently the constraints of the application are not exactly right.

2.4.1 Connection Traps

Given we have two entities that are connected through a sequence of relationships (i.e. more than two). If any instance of these two entities is found not to be in an adjoining relationship instances when it is meant to be present, then we have a *connection trap*.

Let us develop an example. We assume that a team hires many employees, an employee works on a succession of projects, a project has many employees, and it is understood that a team must control a project. A diagram that represents this has three entities, namely **TEAM**, **EMPLOYEE** and **PROJECT**, and two relationships namely a 1(p)-N(t) between **TEAM** and **EMPLOYEE**, and an N(p)-M(p) between **EMPLOYEE** and **PROJECT** – see figure 2.13. A possible connection trap that we need to investigate is whether all instances of **PROJECT**s are related to a **TEAM** as expected. Clearly we have a problem since some instances of **PROJECT** might not have any **EMPLOYEE** assigned to them (there is a partial-partial relationship between **EMPLOYEE** and **PROJECT**) and consequently cannot relate to any **TEAM** instance. This is typically called a *chasm trap* and comes about when a sequence of relationships has

a partial segment. One solution requires us to introduce an explicit relationship between **TEAM** and **PROJECT** and a note that limits this N(p)-M(p) to relationship instances implied by the new **TEAM** and **PROJECT** relationship (i.e. it is a subset).

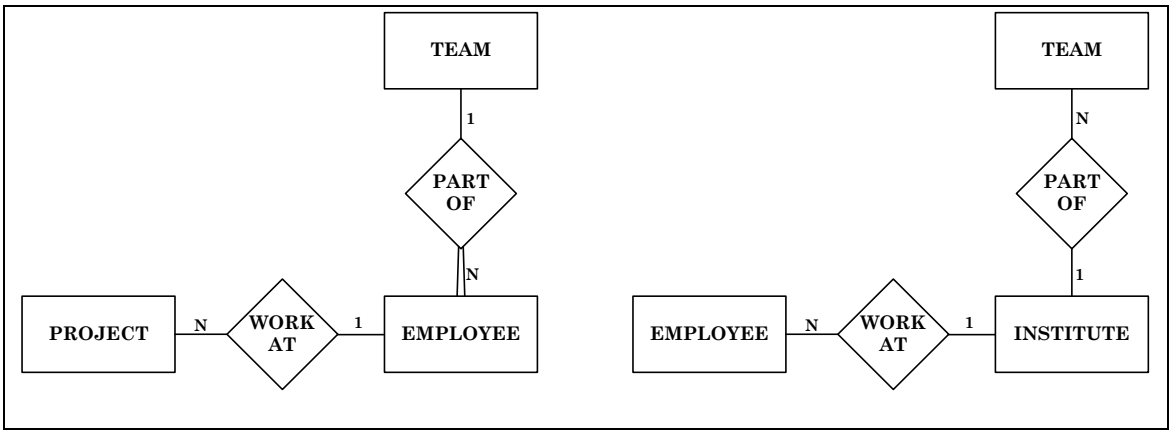


Fig. 2.13 – ERM with a chasm trap (left) and fan trap (right)

Another type of connection trap is the *fan trap*, this occurs when two one-to-many relationships emanate from one entity and an implicit relationship between the latter two is expected. For example let us assume we have **INSTITUTE**s that launch a number of **TEAM**s, and **INSTITUTE**s hire a number of **EMPLOYEE**s – see Fig 2.13. It is of course impossible to determine which **EMPLOYEE**s work on which **TEAM**. To rectify this problem we need to re-order the two 1-N relationships, and specifically state that an **INSTITUTE** launches many **TEAM**s, and each **TEAM** hires many **EMPLOYEE**s.

2.4.2 Many-to-Many Relationships

It is inevitable that one opts for a many-to-many relationship when the constraints and sharing instances seem to be overwhelming a relationship design. Unfortunately this may create a problem as in reality not all relationship instances, although possible in the design, are permissible in an application domain. A simple example is the N-M between students and courses, as in reality there is a generic plan of courses for each degree and only form a "limited" selection does a student choose some units from the whole range (i.e. university wide selection). Another representation of this is the clustering of the relationship instances given a number of students following the same degree program. To solve this too generic N-M one should add notes after exhausting other possible design possibilities. For example add another entity between courses and students such as modules that are a selection of courses a student is allowed to choose from.

2.4.3 Are Three Binary Relationships Equivalent to One Ternary Relationship?

What is a ternary relationship? It is a relationship between three entities and it is used to capture an association between these three entity's instances. The participation level in a ternary relationship is 'one' when an instance is related to one combination of the other two entity's instances; otherwise it is a 'many' participation. Consequently if there is an association between instances from three entities then no three pairs of binary associations can give the same meaning. See figure 2.14 for noting the difference between the meaning of a ternary and binary relationships.

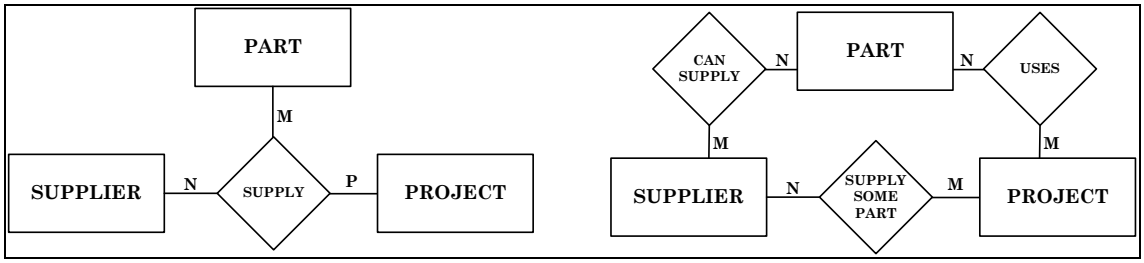


Fig. 2.14 – A ternary relationship is not, in general, reducible to three binary ones.

Any ERM design verification should include a thorough investigation of the necessity of any ternary (or *n*-ary) relationship. It is preferable to have three binary relationships rather than a ternary, implementation wise, but we have just seen that this is not a question of preference but data requirements.

The necessity of a ternary (or *n*-ary) relationship is a problem because some CASE tools and data models only express binary relationships. In these cases the *n*-ary relationship is coerced into an aggregated entity with each of its parts denoting one of the *n*-ary relationships. Furthermore depending on the cardinality of the ternary a set of functional dependencies (a form of integrity constraints) needs to be defined and enforced.

2.4.4 Formalisation of the EERM

In some references, for example in Ng [NGPAU81] and Vossen [VOSSE91], the structure of entities, relationships and a number of constraints (even those found in EERM) are represented through a formal model using set, Cartesian products, and integrity constraints (i.e. first-order predicate logic denial queries).

With formalisation of ERM and including functional dependencies (a form of integrity constraints) one, for example, can easily show that some ternary relationships cannot be represented as three binary relationships. It is sufficient to show that if the combined

primary key sets of the participating entities determine a relationship attribute then there are no three binary relationships that can do so.

Over and above its importance to express the exact meaning of a diagram it also serves us another purpose of great relevance here. Basically these formal structures become our basis for the transformation into another database data model and carry the conceptual structures and constraints onto the database model.

2.5 An Example

A hypothetical database to represent most of the data model and query model examples discussed in this study is presented in this section. This running example describes an organisation entrusted with the categorisation and dissemination of Research and Development efforts. The organisation's scope is in the fields of Information Systems and Telematics being undertaken by a number of research organisations. Both governmental agencies and private organisations are sponsors of these research efforts. This example is from Bertino's exposition found in [BERTI91] and what follows is an informal listing of the system requirements, as simple textual statements.

> A number of projects are currently active and for each project, its unique name, aim and projected total cost are of interest. Also each project assignment undertakes research that has a hardware or software profile, but some projects have aspects from both areas. For hardware projects the details of interest include the devices and the functionality required by the assignment. As for software projects the main interest is in their targeted computer environments and the associated tool-set required during their development.
>
> A number of teams participate in a project's realisation and each team implements a part of the overall project for a predetermined fee. The team's properties of interest include its name (for identification purposes) and its employees. Each team needs to know the total budget derived from the addition of all their participating projects' budget allotments.
>
> A person is an employee of only one team and an employee's pay terms are either on a permanent or on a consultative basis (e.g. per hour basis). A team's employees have a hierarchic placing and (therefore) some employees manage others. Each employee has a distinctive name over the whole scope of the database.

Teams belongs to a research institution (in most cases are part of a University) and receive funds from "outside" sources. Government and industry are the typical outside sources of funding. Each of these sponsors can finance more than one project and more than one team. In each funding instance it is pertinent to determine the exact commitment of a sponsor to each project, categorised by the team's share.

Details of interest about government and companies are their names, main activities and their contact addresses. Some of the address locations, for a number of Government Departments, Institutes and Companies, are the same.

The above requirements are drawn into the EERM given in figure 2.15.

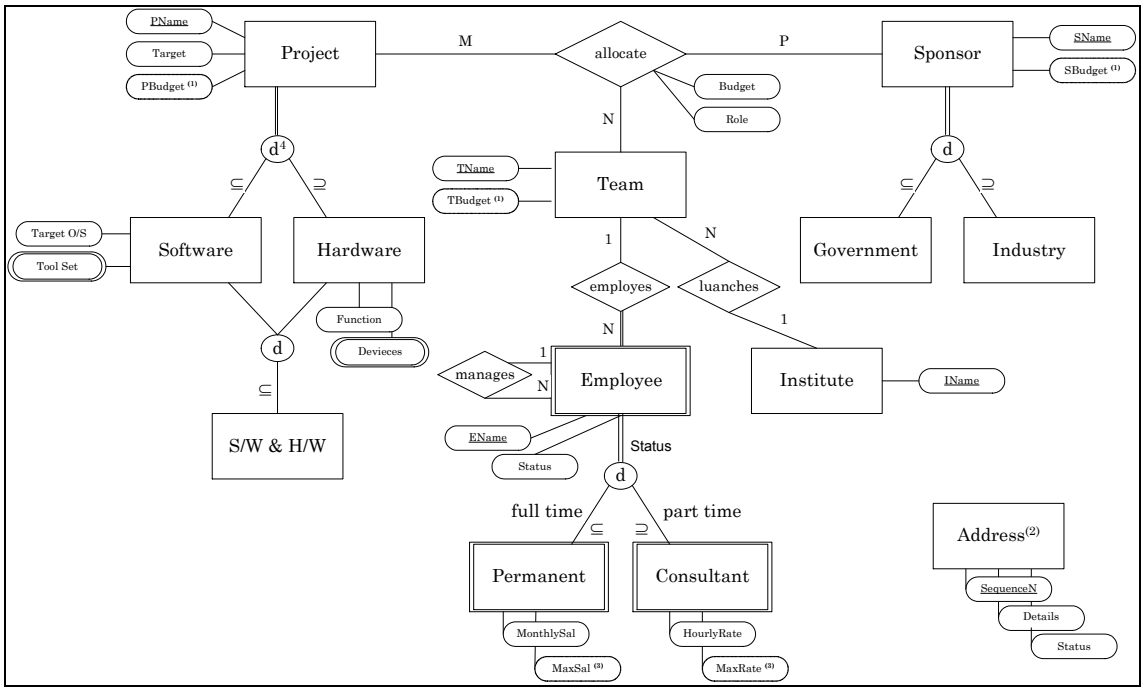


Fig. 2.15 – An example EERM (notes are explained in the following).

Notes to EERM:

(1) The budget value for Project, Sponsor and Team instances is computed by summing all instances' budget contributions given in their respective allocation relationship.

(2) Use the Address instances as part of an aggregation. Relate as independent and shareable.

(3) These are really 'class attributes'; consequently all instances would have the same value.

(4) Should this relationship be changed to an overlap type then the multi-inheritance and entity 'S/W & H/W' are redundant.

2.6 Summary

The assertion that Chen's ERM is a popular top down design language is widely supported; the main justifications being its high-level nature and independence of any database data model. A significant branching from it is found in part of UML. This chapter surveys conceptual modelling through Chen's and other later works in the area.

Diagrams are composed of entities, relationships, and constraints. Later additions to ERM, including sub-classing, categories and aggregation, are called EERM. The introduction of these constructs not only requires malleable structures but also effective mechanisms to describe how the underlying data changes; for example changes to a part-of hierarchy required detailed data-changing procedures.

The automated conversion of an EERM into an object-oriented data model is a priority in this study. The conversion needs to have a high-degree of correctness and completeness. The effects of incorrectness are obvious; on the other hand the effects of incompleteness would imply that the 'missing' artefacts need to be hard wired and consequently disconnected from the DBMS control.

The following chapters deals with a definition of an object-oriented database and data model, and presents and an object-oriented database standard. In chapter nine, where our original work is presented, shows how an EERM model is translated into a standard object-oriented database language. The translation includes entities, attributes, various types of relationships, and constraints (e.g. referential, and primary key set).

Chapter 3

Object-Oriented Paradigm: Object Basics

3 – Object-Oriented Paradigm – Object Basics

The object-oriented paradigm pioneers are *Simula* [DAHLO66] and *Smalltalk* [GOLDB88] and an early seminal paper is of Stefik and Bobrow [STEFI86]. An underlying theme, which attracts attention, is the mashing of structure and behaviour together in an object and availability of a development environment that further aids design and development though a number of features and mechanisms. Currently the most popular programming language must be object-oriented *Java* [GOSLI05].

Object-oriented databases (or *object databases*) are repositories whose schema is defined through object-oriented modelling, which has a basis in the *object-oriented paradigm*. Data modelling languages are no longer abstract; also they are the target language of the conceptual design surveyed in the previous chapter. In this translation the data modelling must implement as much as possible of the design found in the diagram.

A proposal, coming from the Object Database Management Group (ODMG) [CATTE00], of a standard has its own data and query model, and descriptions for latching an object database to object oriented programming language environments; e.g. coupling to Java. The effort has good support and passed a sequence of revisions.

Nonetheless, no "common" object-oriented data model exists in the sense of the relational data model. There seems to be two main reasons for this deficiency. Firstly the basic themes of the object-oriented paradigm are data abstraction through objects, identification, object value, classification, inheritance, and type polymorphism. Although the feel of these themes is quickly grasped, their combination hides a subtle variety of options. Furthermore the options and implementation issues available within the paradigm are staggering. Wegner in [WEGNE89A] identified some 128-design options based on seven paradigm features and states that "some are more interesting than others". Secondly the relational model, together with the relational calculus and algebra, has its formal basis in first order predicate calculus. This logic is understood and accessible. On the other hand, object-oriented themes come from a widespread spectrum of research areas (as varied as computer science, type theory and psychology) – see figure 3.1. Consequently each theme comes with its own theoretical background, requirements, and idiosyncrasies.

This chapter and the consequent two survey the object-oriented themes and object-oriented data modelling with a strong bias for the database perspective. The fundamental theme is the object and other themes include identification, classes, inheritance, and data typing. The chapter leads to the definition of an object-oriented database. Consequently this definition is compared, and used to explain, an object-oriented data model adopted by OMG and ODMG.

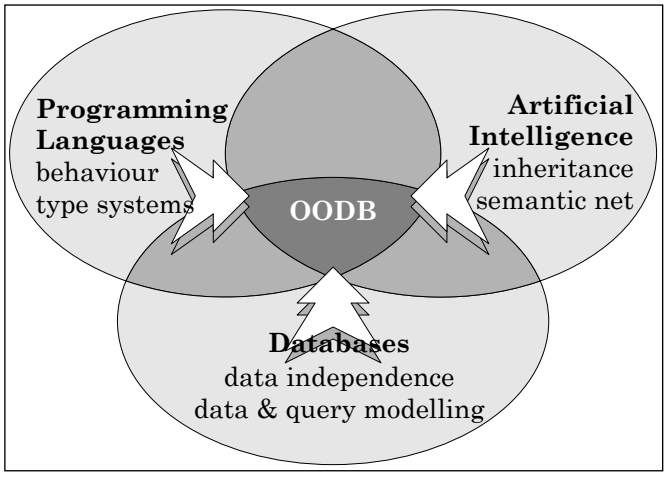


Figure 3.1: Spheres of influence on OODBs development

In most of the object-oriented data models the object is the major construct. The objects model an information system's instances. At a shallow level, an object is a capsule of properties that includes the instance's unique identity, state value and behaviour. An object has a life history: it is created, updated and possibly purged. This locality of object properties is conceptual. Using the consumer and supplier analogy, processing with objects entails consumer objects specifying what a supplier object should do for them but implicitly it is the supplier's prerogative to attempt to match an appropriate behaviour to the consumer's requests.

The aim of this chapter is to expose high-level ideas of encapsulation and then to start building a definition of an object. The basis of an object's value part description is structural and consequently straightforward. Object interaction through message passing invoking methods is also surveyed in this chapter. A graph structure of values is used to depict an object's state. In the later parts, i.e. object identifier section, this graph structure is redefined to include value sharing through identifiers.

3.1 Encapsulation

In object-oriented programming languages *encapsulation* is applicable at an object level and it provides feature locality and supports information hiding [MITCH03]. The *feature's locality* implies a collection of an object's identity, state values and behavioural semantics into a single artefact. The encapsulation feature provides for *information hiding* by having an object declare a number of operations as part of an *external interface* that is available to its clients and another set of operations as part of an *internal operation* that are only for its own use and hidden from any clients. Clearly the external interfaces partially specify the behaviour of an object while the internal operations implement its behaviour.

One of the important mechanisms for data abstraction in an object-oriented paradigm is encapsulation. Encapsulation, in turn, develops structures where modularity can yield better software development by defining the external interfaces through which objects interact. Data abstraction has a strong grounding in computing. For example Simula [DAHLO66] has encapsulation and Parnas' exposition [PARNA72] on data abstraction shows the advantages of modularity in software development.

Other than locality and information hiding, encapsulation favours:

1) An increase in *modularity*, loosely described as "autonomous, coherent and organised in a robust architecture" programming artefacts [MEYER96], yields a dramatic reduction of interdependencies amongst entities.

2) An increase in comprehension capability of developers brought about by examining the external interface of objects.

3) A relatively narrower range of objects is affected by code maintenance brought about on supporting an Information System's evolution.

4) A capability for an object's internal restructuring if its external interface remains intact.

3.1.1 External Interfaces

The external interface is typically described with a set of operations. Each operation has a name and a list of arguments with their corresponding type. Also an indication of the returned object type is given. Note that this technique avoids giving a formal specification of each interface operation; in fact this is only a syntactic rendition of the interface.

3.1.2 Data Independence and Encapsulation

The virtues of data independence in database modelling are well known. So are, at this point, the virtues of encapsulation for application development. Although data independence and encapsulation is not the same thing they are reconcilable with a compromise. In object-oriented data modelling an object's structure is an important part of its definitions. This contrasts with object-oriented programming (especially those with abstract data type genera) where an object's structure is an implementation detail and the public interface provides the main view of the object. A rational explanation is to place an entity's attributes and behaviour as part of an object's external interface and ensure that no data modelling construct assumes anything about an object's internal implementation (e.g. object placement or property's vicinity of other attributes).

A valid reason for wanting to know the structural make-up of an object is to enable querying objects, especially those objects made up of other sub-objects. The resolution of this dilemma follows the above solution of placing this structural profile into the external interface. Nonetheless these external interfaces specifications must remain free of any physical specifications.

3.1.3 Encapsulation and Inheritance

Encapsulation in systems where objects do not share or export their specification is a straightforward mechanism. When an object imports part of its specification from another object (this is a form of specification inheritance) encapsulation is no longer as robust. In this case the specification supplier has constraints on its external interface through the importing objects' dependencies. This is a conflicting situation, specifically how to balance internal re-organisations needs with other objects' interpretations, has long troubled the paradigm (Synder's OOPSLA paper [SNYDE86] is one of the first reports on encapsulation and inheritance in evolving application development). Chapter four on "Classes and Inheritance", considers the consequences and possible solutions to this problem in further detail.

3.2 Objects and Values

In this section we dissect various models that describe object components; e.g. identity and value composition. The build-up of an object composition starts from the simple to the

complex model with logical identity. The following sub-sections introduce some basic notation and sets.

3.2.1 Basic Sets

A *set* is a collection of instances (some of which could be sets) and is denoted by an enumeration of the elements enclosed with curly brackets; namely "{" and "}". Some sets have a name. The *null set* is a special case of a set (because it is a set with no elements) and it is universal. Null sets symbols include "{ }" or ∅.

This subsection is mostly based on Beeri's robust terminology [BERRI90A]. It is assumed that there exist a restricted number of *basic domains*, for example *integers*, *characters* and *Booleans*. Domains need to have their content well known – i.e. universally known. An *atomic value* is an element of a basic domain; for example '9' denotes an instance of the integer's domain. Values are fixed, visible, and are ever present. Each basic domain, $\mathcal{D}_i$, has its own name, $\mathcal{DN}_i$. $\mathcal{D}$ is the disjoint union of all $\mathcal{D}_i$: $\mathcal{D} = \bigcup_i \mathcal{D}_i$.

In the database culture sets are expected to be homogeneous. For example the set of salaries has all its elements pertaining to the domain of integers.

Attributes are named characteristics of structures (or objects). There is a countably infinite number of attribute names in set $\mathcal{A}$, and each instance in this set, denoted by $\mathcal{AN}_i$, has an association to a domain.

A *tuple* is a structure that encloses its attributes (we use the square brackets "[" and "]" to enclose them). A tuple's specification includes a set of attributes name and value pairs. If $a_i \in \mathcal{A}$ and $n > 0$ then $[a_1{:}v_1, \ldots, a_n{:}v_n]$ is an *n-tuple* instance. The "[]" denotes the *null tuple*. Tuple t, for example, with a_j attribute has its value denoted by $t[a_j]$; $t.a_j$ is a misnomer adopted within the database culture.

For a more flowing description the existence of some other sets is given before being explained. The first is the set of class names, $\mathcal{C}$, that denotes a finite number of class names, $\mathcal{CN}_i$. The second is a possibly infinite set of identifiers, $\mathcal{O}$. There is a special identifier, called *null identifier*, which points nowhere. Elements of $\mathcal{O}$ have a number of notational variances; typical examples are id_i and o_i.

The enumeration of the set of structured values without object identifiers, val, depends on $\mathcal{D}$ and a set of structural composition rules (e.g. tuple constructor). Each set of rules describes a particular state structure. In the next sub-sections an overview of three such rule sets is given.

3.2.2 Tuple Structured Values

Structured values in the relational data model are the tuples, representing instances, which relations (i.e. tables) take. The tuples are a subset of the Cartesian product of the relation's schema attribute domains. The tuple and set are *type constructors* and create the structured value schemas. See table 3.1 for the data and type definition of relational values. The relational model's *first normal form* severely limits the structural profile of values in two ways. First, the only valid use of the type constructors is a set following a tuple type constructor application. Second, the attributes may only range over basic domains.

Table 3:1 Tuple Structure Values

Rule Generating Part

The rules that generate the set of relational structure values, val_r, on $\mathcal{D}$ are:

1) All instances of $\mathcal{D}$ are values of val_r. Therefore all atomic values are part of val_r;
2) If $v_1, \ldots, v_n$ are n values and $a_1, \ldots, a_n$ are a distinct sequence of attribute names from $\mathcal{A}$ then $[a_1{:}v_1, \ldots, a_n{:}v_n]$ is a tuple type structure value, val_r.

Typing Part

The world of types is: $\boldsymbol{t} ::= \mathcal{D}$

Type $\boldsymbol{t}$'s interpretation, $\| \boldsymbol{t} \|$, follows:

1) $\| \varnothing \| = \varnothing$
2) $\| \mathcal{D} \| = \mathcal{D}$; and
3) $\| [a_1{:}\mathcal{D}, \ldots, a_k{:}\mathcal{D}] \| = \{ [a_1{:}v_1, \ldots, a_k{:}v_k] \mid v_i \in \| \mathcal{D} \|, i = 1 \ldots k \}$.

3.2.3 Non First Normal Form Structured Values

The first normal form slackening in relational modelling was Makinouchi's proposition in [MAKIN77]. This work led to a more liberal use of the type constructors. The *non-first normal form (NF2)* relations, the name of these new structural profiles, allow the repeated use of the set and tuple constructors (as a whole). NF2 relations, therefore, can have a

relation as an attribute's domain other than a basic domain. The relation's nesting depth can have any depth but fixed at design time. See table 3.2 for the data and type definition of nested relational values. Also, in many NF2 models, another normal form is in force and many call it the *partitioned normal form (PNF)* [ROTHM88]. A nested relation is in PNF if all relation schema attributes are functionally dependent on all the basic domain attributes.

Table 3.2: Nested Relational (NF2) Structure Values

Rule Generating Part

The rules that generate the set of complex structure values, val_{nf}, on $\mathcal{D}$ are:

1) All values of $\mathcal{D}$ are values of val_{nf}. Therefore all atomic values are part of val_{nf};
2) If $v_1, \ldots, v_n$ are values and $a_1, \ldots, a_n$ are a sequence of distinct attribute names from $\mathcal{A}$ then $[a_1{:}v_1, \ldots, a_n{:}v_n]$ is a nested tuple type structure value, val_{nf}.

Typing Part

The world of type is: $\boldsymbol{t} ::= \mathcal{D} \mid [\, a_1{:}\boldsymbol{t}, \ldots, a_k{:}\boldsymbol{t}\,]$

Type $\boldsymbol{t}$'s interpretation, $\|\boldsymbol{t}\|$, follows:

1) $\|\varnothing\| = \varnothing$
2) $\|\mathcal{D}\| = \mathcal{D}$; and
3) $\|[a_1{:}\mathbf{t_1}, \ldots, a_k{:}\mathbf{t_k}]\| = \{ [a_1{:}v_1, \ldots, a_k{:}v_k] \mid v_i \in \|\mathbf{t_i}\|, i = 1 \ldots k \}$.

3.2.4 Complex Structured Values

The data model of Abiteboul and Beeri in [ABITE93A] addressed the set and tuple sequence non-commutatively by advocating for the unrestricted use in the order of these type constructors. Their only restriction on the type constructors' sequence is that the last constructor must be a set. The authors named these structures *Complex values* and in table 3.3 one finds their data and type definition. For an example see figure 3.2. The resultant schemas are acyclic – no part of the schema is defined by itself. Finally, it is easily shown that complex values are generalisations of NF2 and relational values. Also one can map a complex type into a NF2 type by converting a string of type constructors to a sequence of set and tuple pairs – see figure 3.3.

3.2.5 What gains?

What are the advantages attained through the adoption of the nested values and complex values? Firstly we have untied ourselves from the restriction of the relational model's first normal form. Therefore some data access need not require explicit joining of a number of

Table 3.3: Complex Structure Values

Rule Generating Part

The rules that generate the set of complex structure values, val_{cs}, on $\mathcal{D}$ are:

1) All values of $\mathcal{D}$ are values of val_{cs}. Therefore all atomic values are part of val_{cs};
2) If $v_1, \ldots, v_n$ are n values and $a_1, \ldots, a_n$ are distinct sequence of attribute names from $\mathcal{A}$ then $[a_1{:}v_1, \ldots, a_n{:}v_n]$ is a tuple type complex structure value, val_{cs};
3) If $v_1, \ldots, v_n$ are n distinct values then $\{v_1, \ldots, v_n\}$ is a set complex structure value, val_{cs}. (Note the implicit prohibition of a heterogeneous set; this is common in databases).

Typing Part

The world of type is: $\boldsymbol{t} ::= \mathcal{D} \mid [\, a_1{:}\boldsymbol{t}, \ldots, a_k{:}\boldsymbol{t}\,] \mid \{\, \boldsymbol{t}\,\}$

Type $\boldsymbol{t}$'s interpretation, $\| \boldsymbol{t} \|$, follows:

1) $\| \varnothing \| = \varnothing$
2) $\| \mathcal{D} \| = \mathcal{D}$;
3) $\| [a_1{:}\boldsymbol{t}_1, \ldots, a_k{:}\boldsymbol{t}_k\,] \| = \{\, [a_1{:}v_1, \ldots, a_k{:}v_k\,] \mid v_i \in \| \boldsymbol{t}_i \|,\ i = 1 \ldots k \}$; and
4) $\| \{\, \boldsymbol{t}\,\} \| = \{\, \{ v_1, \ldots, v_m \} \mid v_i \in \| \boldsymbol{t} \|,\ i = 1 \ldots m \}$.

relations to access it. Secondly, although not exemplified here, there are a number of reported algebraic and calculus based languages that manipulate these complex structures. Unfortunately these complex structures aren't friendly to instance sharing; this is addressed with the interleaving of object identifiers and complex structures in objects.

Figure 3.2: Complex Values Relation a) Structure, b) Instance and c) Type Tree

a) (note attributes in angle brackets (e.g. members) take a set of values.)

TEAM

ref	desc	manager		< members >			
		ref	name	ref	name	< skills >	< prev_teams >
						skills	< teams >
							members_ref

b)

TEAM

ref	desc	manager		< members >			
10	R&D	ref	name	ref	name	< skills >	< prev_teams >
		101	JADE	101	JADE	skills	< teams >
						DOC	members_ref
						PLAN	101
						DESIGN	105
							members_ref
							101
							109
				ref	name	< skills >	< prev_teams >
				103	JAKE	skills	< teams >
						DESIGN	members_ref
						ANALY	103
						INFO.RET	106
ref	desc	manager		< members >			
20	SND	ref	name	ref	name	< skills >	< prev_teams >
		110	JAM	110	JAM	skills	< teams >
						DOC	members_ref
						PLAN	110
						PRESENT	119
							145
							members_ref
							110
				ref	name	< skills >	< prev_teams >
				113	JOY	skills	< teams >
						PROG	members_ref
						DESIGN	133
						CASE	113

c)

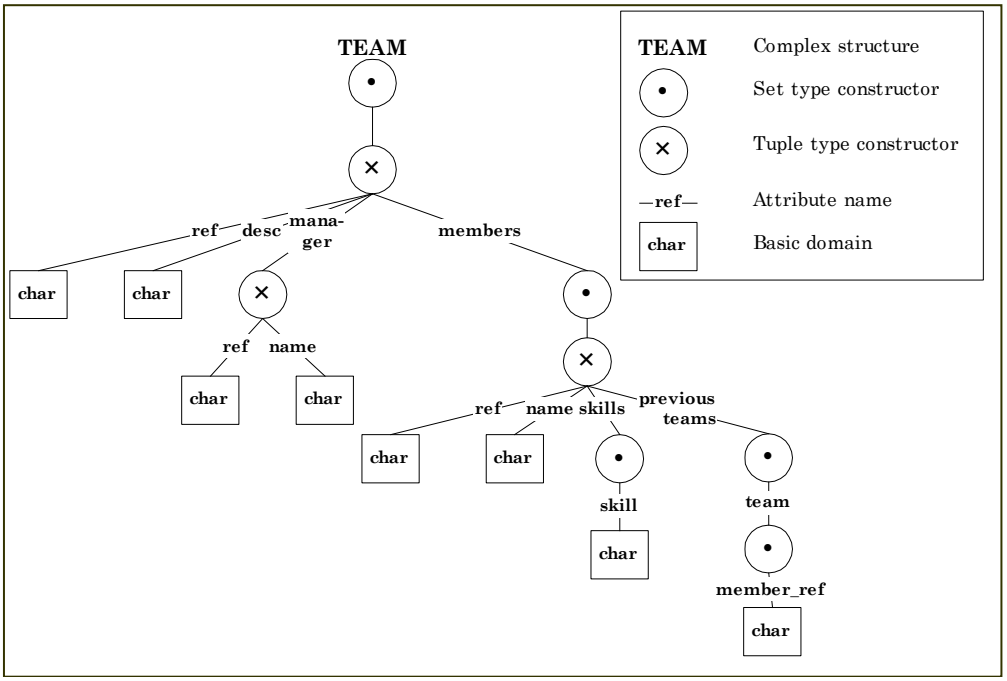
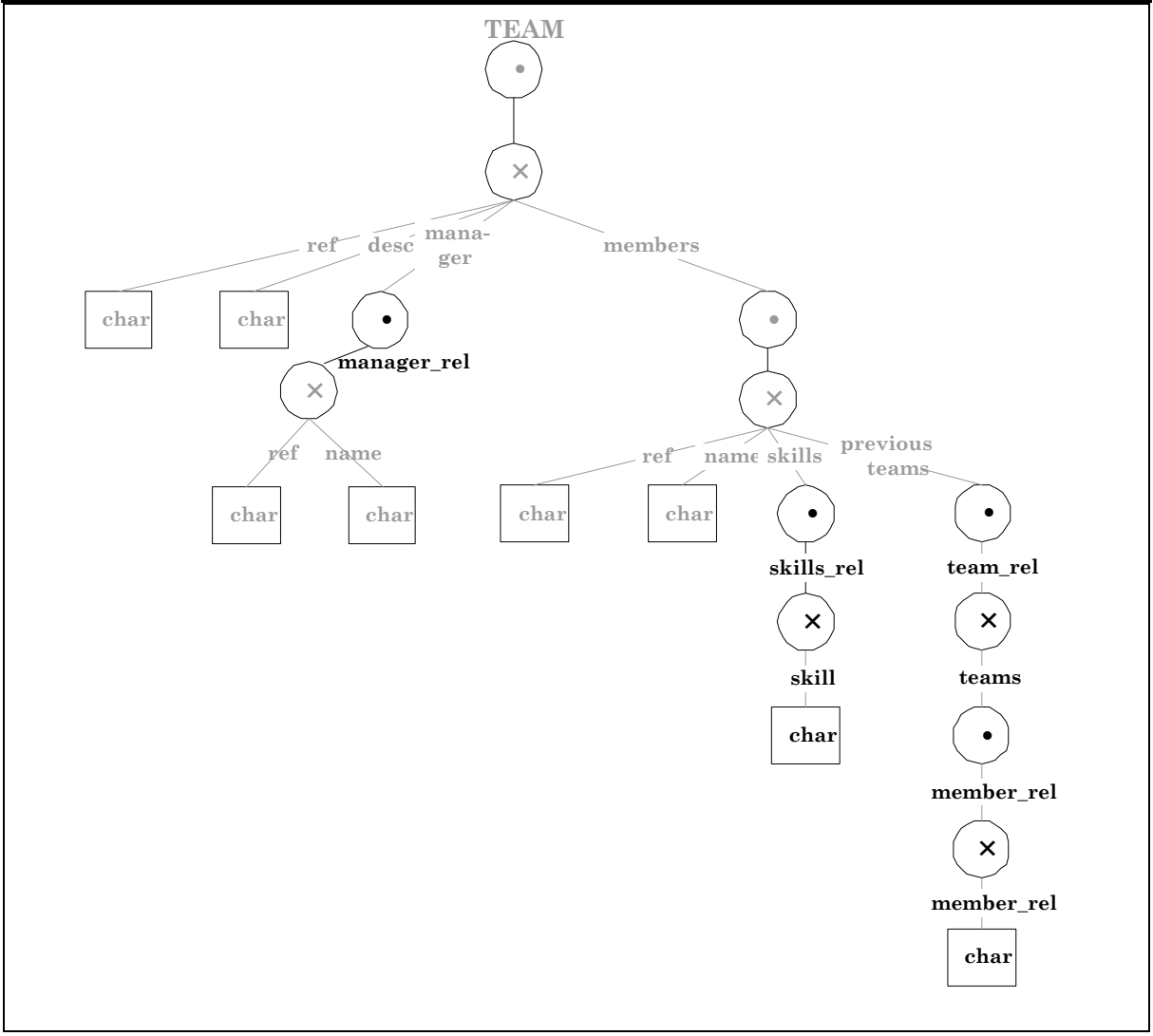


Figure 3.3: Nested Relational Structure (2NF) Type Equivalent to the Complex Value Structure type of Figure 3.2 (c). The darkened part of the figure represent the changes introduced to match the structures.



3.3 Object Identity

Entities and instances that are of interest to an information system must be identifiable and reachable, and consequently identification is part of many conceptual data models. In the database culture there is a strong notion of keys for identification of instances.

In the object-oriented paradigm a key is called an *identifier* and each object has one. An identifier is a handle through which other objects refer to it; the seminal papers on this theme are [KHOSH86] and [ABITE89B]. Typically this identifier is 1) *unique*, it distinguishes its holder from every other object within a collection, 2) *immutable* throughout the object's life span, 3) an *internal* property of an object and therefore is not usually seen or useable by

end users, and 4) its value has nothing to do with its object properties. Furthermore in object databases Atkinson and Morrison [ATKIN95] assert that other than identification issues one also needs to correctly map application domain instances to corresponding repository objects.

When an object is created, called *instantiation*, an identifier is created for it. In any object collection there is a huge quantity of un-instantiated objects; for example integers being values of a basic domain. At this point the subtle issue of what is a value and what is an object is reinforced with the help of object-identifiers (as argument already introduced earlier in section 3.2). The general consensus documented by Beeri [BERRI90A] is to make the following distinctions: elements of the basic domains are "universal" abstractions and ever present, while elements of an information system's entities are "local" abstractions, created on *ad hoc* basis and have a lifetime. A technical refinement to this distinction is that; the former carry their own information and therefore identify themselves; while the latter carry an object's information content that includes relationships to values and to other objects represented by identifiers.

The paradigm offers object access methods based on *navigation* but must not exclude access based on *values*. This contrasts sharply with the access methods found in most programming languages and in relational databases. The following two examples explain common pitfalls in either access method:

1) In programming languages, a memory address pointer (that is *identification by addressing*) facilitates access to a variable's state value and the address is dependent on the executing environment rather than on the object it identifies. A problem may arise when having a number of addresses depicting "different" objects when they are identical. For example, a certain employee satisfies a query (return an employee name (a non-distinctive property) who earns more than £35,000) and he is bound to the variable `WELL_PAID_EMP`, contemporaneously the same employee satisfies another query (return an employee name with a managerial job status and who is working in London) where he is bound to the variable `LONDON_MANAGER`. These two variables are accessible through different paths (for example by pointer re-direction) and for the programming language to support the equality semantics it needs specialised constructs that can ascertain the equality of variables

`WELL_PAID_EMP` and `LONDON_MANAGER` through some pointer chasing mechanism (e.g. *dereferencing*).

2) The relational model introduces the notion of a *value-based key*. Furthermore the relational key is unique within the scope of one table. A marked confusion arises with value based keys because of the differing concepts of value and identity. A serious drawback of this double usage is that the values of the key attributes cannot change without affecting the referential integrity of the database. In some of the world's societies, for example, when a female marries she has to forsake her family's surname for that of her spouse. If the key attribute set contains the family's surname then a change in that value causes an introduction of a second entity (with the new surname) for the same person. To address this in value-based keys, it is a common practice to introduce an "artificial" attribute (e.g. employee number) that has no semantic meaning. Nonetheless this scheme is an effective one, and ironically takes the role of a value-based pointer due to the attributes lack of semantic meaning.

In some pre-relational data models the use of pointers for linking one record to another resembles the paradigm's identifiers. This resemblance led to numerous, and rather loud criticism of identifiers in early object-oriented data models in that they re-introduce pointer chasing present in the traditional models. Furthermore Ullman in [ULLMA87] categorically states that "object-identity does not mesh well with declarativeness". Insisting on the conceptual nature of the object-identifiers and leaving the physical pragmatics as an implementation detail avoids this polemic.

The identification property in an object-oriented data model offers the implicit support of object *equality* and *sharing*. Through its identifier an object is identifiable from any other object and furthermore the identifier is independent of its values, location (in main or secondary store), and addressability. By assigning an identifier to an object's state, sharing of objects throughout the collection is possible and is independent from the assigned values. This sharing introduces the possibility of building cyclic graphs in an object collection. Specifically a number of objects can share an object (or refer to it) without the latter being replicated.

Three binary predicates, IDENTITY, SHALLOW_EQUALITY and DEEP_EQUALITY, are useful to test for the equality of two objects and their level of sharing – this takes care of the identification by addressing issue previously presented. The predicate IDENTICAL evaluates to true when the two objects have the same identifier. SHALLOW_EQUALITY imply that the two objects have equal values and objects for their properties. DEEP_EQUALITY is satisfied if each object's state has equal values when all references are recursively "folded out" by the values they represent. Two identical objects are also shallow equal. Two shallow equal objects are also deep equal. But generally two deep equal objects are not shallow equal and two shallow equal objects are not identical.

A formal treatment of deep equality in the presence of cyclic graphs (and consequently the possibility of infinite trees) is found in Abiteboul's paper [ABITE95B]. Different copying against sharing operators are also required, such as shallow and deep copying, as first found in Smalltalk-80 [GOLDB88] and in Common Lisp Object System (CLOS) [BOBRO86]. Nonetheless it is known from [BEERI99] that, in general, identifiers alone in cyclic structure are not sufficient to distinguish two objects.

3.3.1 Logical aspects of identification

Modelling identity in a logic-based framework has its own considerations. A thorny problem of identification in logic systems is the generation of new identifiers for derived facts. Chapter six has two sections, i.e. on F-Logic [KIFER95] and Flora2 [YANGG08] that addresses this point.

To achieve rendition of identification, logic systems name a specific interpretation for each basic domain. These basic domains have an implied interpretation rather than a rule definition. Consequently each element of the basic values pertains to *interpreted domains* and each instantiated object pertains to *the un-interpreted domain*. Access to values and objects in a logical system are through the ground syntactic terms of the query language (specifically the *Herbrand Universe).*

3.3.2. Complex Value Structure and Nested Relational with Identifiers

Tables 3.4 and 3.5 have the redefinition of a complex and nested relational value with identifiers and figure 3.4 is an example. For a start, an object database schema $\mathcal{S}$ is a pair of $(\mathcal{C}, \mathcal{T})$ where $\mathcal{C}$ is set of classes, $\mathcal{T}$ is the application of $\mathcal{C}$ in types (see type definition part in

figure 3.4). An instance I of the database schema S is a pair too with the following assignments: an assignment π of classes to each of the class instances, and assignment ν between the parts of an object to their value.

Table 3.4: Object Values (Complex Structure Values with Object Identifiers)

Rule Generating Part

The rules that generate the set of object values, *obj*, on $\mathcal{D}$ and O are:

1) All values of $\mathcal{D}$ and O are values of *obj*;
2) If $v_1, \ldots, v_n$ are n object values and $a_1, \ldots, a_n$ are a distinct sequence of attribute names from $\mathcal{A}$ then $[a_1{:}v_1, \ldots, a_n{:}v_n]$ is a tuple type object value, *obj*;
3) If $v_1, \ldots, v_n$ are n distinct object values then $\{v_1, \ldots, v_n\}$ is a set object value, *obj*.
4) All references are associated with instances of a class.

Typing Part

The world of ***TYPE*** is: $\boldsymbol{t} ::= \varnothing \mid \mathcal{D} \mid C \mid [\, a_1{:}\boldsymbol{t}, \ldots, a_k{:}\boldsymbol{t}\,] \mid \{\, \boldsymbol{t}\,\}$

Type $\boldsymbol{t}$'s interpretation, given a certain mapping of identifiers - π, $\| \boldsymbol{t} \|_\pi$, follows:

1) $\| \varnothing \|_\pi = \varnothing$;
2) $\| \mathcal{D}_\pi \| = \mathcal{D}$;
3) $\forall\, C, C \in C, \| C \|_\pi = \pi(C)$;
4) $\| [a_1{:}\boldsymbol{t}_1, \ldots, a_k{:}\boldsymbol{t}_k\,] \|_\pi = \{\, [a_1{:}v_1, \ldots, a_k{:}v_k\,] \mid v_i \in \| \boldsymbol{t}_i \|_\pi, i = 1, \ldots, k \,\}$; and
5) $\| \{\, \boldsymbol{t}\,\} \|_\pi = \{\, \{\, v_1, \ldots, v_m\,\} \mid v_i \in \| \boldsymbol{t} \|_\pi, i = 1, \ldots, m \,\}$.

Note: In 5 of the above a type determines the structure of the values

An object, *o*, up to this point, is a triplet between its identifier, value (either complex or nested relational) and an instantiating class.

```
o = (id, obj, class)          Remark: write o.id object id
```

The meaning of equality, shallow and deep equality is expressed with the following relationships:

```
Remark: assume we want to relate two objects o1 and o2

Remark: object equality

iff ( o1.id == o2.id ) then object_equality( o1, o2 ) holds.

Remark: shallow equality

iff( o1.id != o2.id & o1.obj == o2.obj )  then shallow_equality( o1, o2 ) holds.

Remark: deep equality
Remark:   assume the presence of a function, obj_to_val, that recursively
          replaces all ids in o.obj with basic values
iff ( o1.id != o2.id & o1.obj != o2.obj &
      obj_to_val( o1.obj ) == obj_to_val( o2.obj ) )
then deep_equality( o1, o2 ) holds
```

3.3.3 Identifiers and Referential Integrity

Given the ability to discriminate objects based on their conceptual identifier (and for simplicity we disallow synonyms), postulates on the consequent requirements for a *referentially consistent* collection of objects follow. These two assertions are due to Khoshafian and Copeland [KHOSH86] and must hold for an object collection. The *unique identifier* assumption maintains the uniqueness of the object identifiers in a collection, while the *no dangling identifier* specifies that there can be no reference to an object that does not exist.

Table 3.5: Nested Relational (NF2) Structure Values with Object Identifiers

Rule Generating Part

The rules that generate the set of complex structure values, obj_{nf}, on $\mathcal{D}$ and O are:

1) All values of $\mathcal{D}$ and O are values of obj_{nf}. Therefore all atomic values are part of obj_{nf};
2) If $v_1, \ldots, v_n$ are n values and $a_1, \ldots, a_n$ are a distinct sequence of attribute names from $\mathcal{A}$ then $[a_1{:}v_1, \ldots, a_n{:}v_n]$ is a nested tuple type structure value, obj_{nf}.

Typing Part

The world of type is: $\boldsymbol{t} ::= \varnothing \mid \mathcal{D} \mid C \mid [\, a_1{:}\boldsymbol{t}, \ldots, a_k{:}\boldsymbol{t}\,]$

Type $\boldsymbol{t}$'s interpretation, given a certain mapping of identifiers - π, $\|\boldsymbol{t}\|_\pi$, follows:

1) $\| \varnothing \|_\pi = \varnothing$;
2) $\| \mathcal{D}_\pi \| = \mathcal{D}$;
3) $\forall\, C, C \in C, \|C\|_\pi = \pi(C)$;
4) $\|[a_1{:}\boldsymbol{t}_1, \ldots, a_k{:}\boldsymbol{t}_k]\|_\pi = \{\, [a_1{:}v_1, \ldots, a_k{:}v_k] \mid v_i \in \|\boldsymbol{t}_i\|_\pi, i = 1, \ldots, k \,\}$.

These two assumptions are integrated into the object values with identity by introducing the following integrity constraints (integrity constraints are explained in detail in chapter seven):

Remark : all classes have their instance identifiers in the collection and each value part of each object respects their value structure (including identifiers that must be in the collection)

$$\forall C \in \mathbf{C}, \forall \mathrm{o} \in \pi(C), \nu(\mathrm{o}) \in \|\mathbf{T}(C)\|_\pi$$

3.3.4 Identification's Pragmatics

The conceptual object identifier has a physical, or a system level, counterpart. At this latter level the identifier is an address to a memory storage space. The principal engineering issues for physical identifiers are their generation and the efficiency of operations they participate in. These operations are basically the retrieval of the

referenced object and possibly retrieval of other objects referenced in a target object; consequently a good part of a DBMS performance depends on efficient implementation of the referencing mechanisms *vis-a-vis* expensive disk access requirements. Two common mechanisms follow.

In the first mechanism a function, that guarantees the uniqueness of the spawned identifier, bestows an identifier to an object on instantiation. The next step involves placing the object's state into a storage space slot. Finally, add to an index entry that associates the identifier to its physical location. Some object DBMS refine this by augmenting the spawned identifier with typed information (for example the class instance that created the object). Typically, with extensible hashing techniques, an object access takes one or two disk page reads. POSTGRES [STONE90A] is one of the earlier implementations to adopt this scheme.

In the second mechanism the identifier is actually a physical address of the object in the storage space. It is a structure of values (for example a volume and disk page). There are two main problems here; this regime loses its conceptual nature previously advocated, and an object that has to move into another storage location must leave, in its original address, a pointer to the new location. This has two effects on the DBMS performance and whose severity depends on this redirection frequency of occurrence: 1) access is no longer in a single data access path but in two; 2) storage page fragmentation becomes more difficult to control. The ONTOS [ANDRE90] implementation advocated this structural scheme.

A final point about the implementation details of object's identifier concerns the preservation of a collection's referential consistency. In a *strong identity* regime any operation over the collection must preserve the assumptions at all costs. For example, an object stays in existence for as long as any reference to it is present: that is no explicit purging operation is available and hence garbage collection must be present. Strong identity can cause problems associated with *logical pinning* too (inaccessible objects continue to reside in a collection).

Goldbreg and Robson [GOLDB88] suggest ignoring these rules to allow a level of *weak identity* (with explicit object destruction being supported); this is acceptable only if an

adequate exception-handling procedure exists to support the situation when an object reference is no longer satisfied by an object in the collection (called *dangling reference*).

Figure 3.4: Nested Relational Structures with Identifiers Example

The database schema is:

S_1 = ({ **TEAM**, **PERSON** }, T)

Remark: object identifiers are in italic and arbitrary start with the letter O

Remark: classes are written in bold (e.g. **PERSON**)

```
T ( TEAM )   = [ DESC : CHAR, MANAGER : PERSON, MEMBERS: { [ MEMBER:PERSON ] } ]
T ( PERSON ) = [ NAME : CHAR, SKILLS : { [ SKILL:CHAR ] },
                 PREV_TEAM_MEMB : { [ PREV_TEAM:{ [ MEMBER:PERSON ] } ] } ]
```

A database instance I_{s1} that is based on S_1 is:

I_{s1} = (π, ν)

Remark: a mapping from classes to its extent in object identifiers

```
π ( TEAM )   = { O10 , O20 }
π ( PERSON ) = { O101 , O103 , O105 , O106 , O109 , O110 , O113 }
```

Remark: partial function from an object identifier to its value (in obj)

```
ν ( O10 )  = [ DESC : 'R&D' , MANAGER : O101 ,
               MEMBERS : { [ MEMBER: O101 ] , [ MEMBER: O103 ] } ]
ν ( O20 )  = [ DESC :'SND' , MANAGER : O110 ,
               MEMBERS : { [ MEMBER: O110 ] , [ MEMBER: O113 ] } ]
ν ( O101 ) = [ NAME : 'JADE' ,
               SKILLS : { [ SKILL : 'DOC' ] , [ SKILL : 'DESIGN' ] } ,
               PREV_TEAM_MEMB : {
                   [ PREV_TEAM { [ MEMBER: O101 ] , [ MEMBER: O105 ] } ],
                   [ PREV_TEAM { [ MEMBER: O101 ] , [ MEMBER: O109 ] } ] } ]
ETC
```

3.4 Message Passing

The intra-object communication aspect of most object-oriented systems is taken care through a *message passing* metaphor. This metaphor is not exclusive to object-orientation even though many see the influence of early object-orientation on the pioneering Actor [HEWIT73] model. A communication starts by identifying an object and passing to it a message together with a number of arguments. The receiving object becomes active and reacts by executing an operation it associates this message; this operation is called a

method. It is imperative to emphasise that the operation an object executes is part of its own definition and not a part of the message transmission.

For example the message expression `(2 PLUS 4)` reads as follows; ask object denoted by "two", an integer, to execute its plus operation with object "four" as an argument. For this message, object "two" binds to his method `PLUS` and pops the object "six". If the message expression is part of an assignment, `SUM <- (2 PLUS 4)` is an example, then the variable `SUM` is assigned to the object "six" after executing the expression.

Message passing provides for blocked asynchronous inter-object communication. The most common mechanism is for the receiving object to pop the expression's resultant object on to the environment. An object collection is computationally passive; that is, no object is executing a method. A collection's *environment*, on the other hand, implies that there is a computational process-taking place; it is the usual case that there is at least one active object at a point in time The asynchronicity aspect of message passing comes about by the transfer of the current object status from one object to another in a given environment. In the last example, when "two" is the current active object it is executing the message plus and the assignment is blocked. On "two" finishing the plus computation, the object `SUM` becomes active while "two" recedes into being an inactive object.

3.4.1 The Message Passing and Method Determination Mechanisms

The message-passing paradigm hides a number of elaborate and subtle mechanisms. The first mechanism accepts, or otherwise, messages sent to it. The messages that an object accepts are those included in its external interface. To each external interface there is a separate mechanism that attaches a method (or an attribute) definition to it. (The latter mechanism enforces a partial ordering of methods from local and inherited definitions based on their vicinity to the receiving object – details of which are in the "classes and inheritance" chapter.) The closest method *overrides* all other applicable methods found in this partial ordering. The lack of prior knowledge, even to the receiving object, of which method to execute for satisfying a message leads to a *late binding* practice. A message and an object's external interface mismatch compel the receiving object to returns a *'message not understood' error message* to the sending object.

A method's computation sometimes entails invoking other methods attached to the current active object; that is an object sends a message to itself. A pseudo name for the current active object, like *self* in Smalltalk, realises this mechanism. The self name appears in a method's specification. An effective example of the self pseudo name's use is the following Smalltalk [GOLDB88] implementation of the factorial function:

```
factorial
     self    = 0
             ifTrue:         [^1].
     self    < 0
             ifTrue:         [self error: 'factorial invalid']
             ifFalse:        [^self * (self - 1) factorial]
```

Another pseudo name is super (again using Smalltalk's terminology). The use of *super* in method implementation forces the receiving object to bind not with the most local method but the next one along the partial ordering of methods associated with the receiving object (this is explained further in the "classes and inheritance" chapter). The most common use of super is when a method definition involves an overridden method. This is an important and effective code re-use facility. It is sometimes known as *incremental behaviour specification*.

The determination of which method to execute when an object receives a message depends on two mechanisms: firstly on the external interface and secondly on the method look-up through the local and inherited method implementation. These two mechanisms effectively isolate the scope of the methods relative to a receiving object.

It is common that a method name is ubiquitous in objects but each invocation entails separate semantics. This methodology is called *method name overloading*. A text book example of overloading is the `PRINT` method example. Most objects require a different implementation and consequently the name `PRINT` is overloaded.

Method name overloading occurs throughout the collection while method name overriding is a selection mechanism applicable within the scope of the receiving object's own and inherited definitions.

3.4.2 Message Passing and Function Calls

Structurally the syntax for function calls and message passing is easily reconcilable: a function call takes the form of `FUNC(`ARG_1`, ... , `ARG_N`)`; while a message passing takes the form of `RECEIVER_OID MSG `ARG_1` ... `ARG_N.

Assume the presence of a set of method names, $\mathcal{M}$. A method expression for $\mathcal{Mn}$ follows, where $\mathcal{Mn}$ is a member of $\mathcal{M}$, $\mathcal{Cn}$ is the class with which the method is associated, and $\mathcal{Tn}$ are types for arguments and return type.

$$Mn : Cn \times Tn_1 \times Tn_2 \times ... \times Tn_n \rightarrow Tn$$

The semantic contrasts between these two include late binding and method name overloading. Also message passing requires the presence of a privileged and ever present argument; which is the receiving object.

There are two important issues here – look-up performance and execution safety. For late binding it is imperative that the look-up operation is lightweight and comparable to looking up a function; the works by Odersky and Walder have received attention – i.e. in [ODERS97]. Also in late binding the compiler can catch many improper message calls and arguments; but it is not absolute. The many is becoming most in just in time compilers developed from techniques introduced in *Self* by Smith and Ungar [SMITH95]. This is revisited in data typing chapter (i.e. Classes and Inheritance).

3.4.3 Method code

In object databases a method implementation is often coded with an object-oriented language.

Although novel, the use of methods in a data model has drastic consequences. For example if a method's code is changing the underlying state of the database then if its computation depends on the state the method is seeing then the method invocation becomes potentially *un-safe* in the sense that it might never end. Method implementations that do not change the under-lying database state are said to be *side-effect free*. Some method's implementation that do change the state are not necessary un-safe. Deciding which methods are side-effect free is generally a NP-hard problem (this follows from the requirement that decision process need to simulate the semantics of the method's implementation).

3.4.4 Concurrency and Message Passing

A high-level characterisation of concurrency in computing is a number of independent activities executing and correctly sharing a computational environment. These independent activities require management and intercommunication.

The object-oriented paradigm's message passing combines two things: the first is activity transfer between the recipient and the sender; the second is limited synchronisation. These alone do not provide a basis for a concurrent framework. The fact that Simula and Smalltalk simulate concurrent systems is not the point. Nonetheless, an object's feature locality helps with realising concurrency because of the reduction inter-object communication. This feature translates into a simplified placement and separation of objects at an implementation level.

3.4.5 Message Passing in Object-Oriented Databases

The Orion object-oriented databases has message passing for both object definition and manipulation. The example is adopted from a paper describing Orion [KIMWO90D].

```
Remark:  the following examples are based on the Orion OODB prototype
         Orion's message passing syntax is
         ( selector receiver [ arg1 arg2 arg3 … ] )

Remark:  where selector is the message and receiver is the object
         whose class definition is requires
         (note Orion's syntax is not consistent here!)

( make-class TEAM
    :attributes (( desc    :domain STRING )
                 ( manager :domain PERSON )
                 ( members :domain ( set-of PERSON ))))

Remark:  object / instances definition
( make TEAM :desc 'R&D' :manager ( make PERSON :name 'Jade' ) )

Remark:  return a set of team objects with Jade as its manager
( select TEAM ( manager name = 'Jade' ))
```

3.5 Summary

This chapter's main aims are two: enumerate the better and adequate sources that describe basic object-oriented features; accentuate and build bridges between these basic features and database design requirements. The basics covered in this chapter are encapsulation, object identity, object values with identity, and message passing. These are the basis for other object-oriented themes and variations described in the next two chapters (i.e. four and five).

Encapsulation offer the object-oriented paradigm adopters feature locality and information hiding. We have also reported on how encapsulation and data independence overlap as there are possible conflicts with inheritance mechanism due to other dependences found within inheriting objects.

Herein we gave a progression on how to build an object's value; we started with tuple and continued with nested values and complex value and finally augment with logical identifier to enable better objects sharing their values. Object identity, at logical level, is a unique and immutable identifier. Also we are able to compare an object's values at various levels: identifier, shallow, and deep. A formal treatment of an object is given.

In the chapter four these features are used to describe more involved object-oriented themes like classification; classification is a very important database modelling theme. Also the principles of inheritance and its pragmatics are given. Finally data typing checking and inference in an object-oriented environment is covered too.

Chapter 4

Object-Oriented Paradigm: Classification and Inheritance

4 – Object-Oriented Paradigm –

Classification and Inheritance

The concepts of object and identification are used to build the other more involved concepts of the object-oriented paradigm. In this chapter the main concepts are classes and various modes of classification for data modelling, inheritance, and data type checking and inference in object-oriented environments.

Classification is fundamental for data modelling and data typing. Also inheritance, attached to classes, aids in data modelling and application development. The need for a flexible data typing regime is a must and many of the advance features had the paradigm as a test bed. Research up to the 2000 is adequate to cover our data type requirements.

Once these features are synthesised and focused for database design this chapter leads to a description of an object-oriented schema (found in the next chapter).

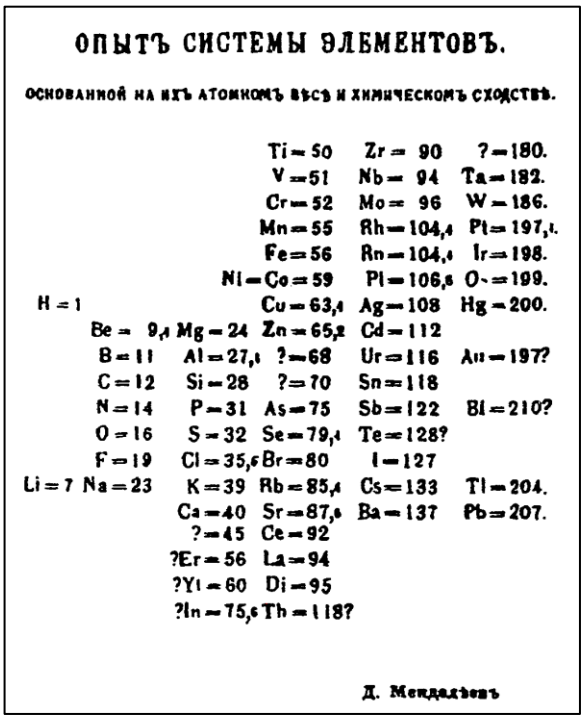
4.1 Classes and Classification

Other than identity and state an object has an association with the artefact that created it; the latter is typically called a *class* while the association is called an *instance of* relationship. Classes collate, or compose in data modelling terminology, structural and behavioural properties of objects that are their instances. The following is an example of a class definition in Orion [KIMWO90D] for noble gases and consequently all of its instances has, for example, an atomic weight property. In this example it is possible to see how some 'attributes' can denote values by a reference to other objects (e.g. supplier refers to class **SUPPLIER** and consequently objects are assigned an identifier from its instances).

```
Remark:  a class definition in Orion's OODB prototype syntax
(make-class  NOBLE_ELEMENT
    :superclasses ( ELEMENT)
    :attributes   ((atomic_weight :domain REAL)
                   (state_at_15c  :domain STRING)
                   (supplier      :domain SUPPLIER))
    :methods      ( purchase_order
                    clear_stock_level))
```

In a wide variety of scientific systems a *classification* is the division of a domain into partitions and forms a basic structural representation of the subject domain. In most cases the instance-of relationship is semantic in nature and few follow a rule to determine

membership. Mendeleev's periodic table for chemical properties of elements is an excellent example of a lateral classification; every element is a member of one element class (see figure 4.1). For example, in the case of the periodic table, Helium and Argon are instances of the **NOBLE_ELEMENT** class. The collection of all instances of a class is called its *extent* (e.g. the extent of **NOBLE_ELEMENT** includes Helium, Argon, etc.). Classification is also a basis on which a number of other conceptual abstractions, like specialisation and aggregated composition mentioned in the conceptual design chapter, are built.



ОПЫТЪ СИСТЕМЫ ЭЛЕМЕНТОВЪ.

ОСНОВАННОЙ НА ИХЪ АТОМНОМЪ ВѢСѢ И ХИМИЧЕСКОМЪ СХОДСТВѢ.

		Ti = 50	Zr = 90	? = 180.
		V = 51	Nb = 94	Ta = 182.
		Cr = 52	Mo = 96	W = 186.
		Mn = 55	Rh = 104,4	Pt = 197,1.
		Fe = 56	Ru = 104,4	Ir = 198.
		Ni = Co = 59	Pl = 106,6	O = 199.
H = 1		Cu = 63,4	Ag = 108	Hg = 200.
	Be = 9,4	Mg = 24	Zn = 65,2	Cd = 112	
	B = 11	Al = 27,4	? = 68	Ur = 116	Au = 197?
	C = 12	Si = 28	? = 70	Sn = 118	
	N = 14	P = 31	As = 75	Sb = 122	Bi = 210?
	O = 16	S = 32	Se = 79,4	Te = 128?	
	F = 19	Cl = 35,6	Br = 80	I = 127	
Li = 7	Na = 23	K = 39	Rb = 85,4	Cs = 133	Tl = 204.
		Ca = 40	Sr = 87,6	Ba = 137	Pb = 207.
		? = 45	Ce = 92		
		?Er = 56	La = 94		
		?Yt = 60	Di = 95		
		?In = 75,6	Th = 118?		

Д. Менделѣевъ

Figure 4.1 – Mendeleev's periodic table (1869) – the "?" named objects were then unknown but predicted elements HTTP://EN.WIKIPEDIA.ORG/WIKI/DMITRI_MENDELEEV

A basic question arises: is a class artefact an object too? We believe that this reification is of benefit to the paradigm and database culture. Accepting classes on par with instances (e.g. sometimes referred to as *object homogeneity*) has the following effects: classes have their own encapsulation; relationships with other like objects (e.g. classes); relationship with their own extents and act as the interface to their instances. Some sources call artefacts that can instantiate classes '*meta-classes'*. Attributes that are part of a class composition or represent a collective property of all its instances are called *class attributes*. In some systems *static methods* have a similar nature.

It is pertinent to state, at this point, that the ERM's entities and relationships are reconcilable with the paradigm class artefact. So are the EERM sub-setting and aggregation artefacts.

4.1.1 Class and Generalisation Abstraction

Another classification comes from biological sciences and where classification of the animal kingdom is given through a hierarchic representation; its origin is attributed to Linnaeus (HTTP://EN.WIKIPEDIA.ORG/WIKI/CARL_LINNAEUS).

Through this *class hierarchy* we can also compare instances, if comparable, of two sets. If these sets are converted into class abstractions then a relationship between classes is built. This is the *ISA* relationship on classes. We say that the class **PRIMATE** *ISA* **MAMMAL** and the class **HUMAN** *ISA* **PRIMATE**. The *ISA* relationship between classes is in general reflexive, transitive, and anti-symmetric, and consequently a *partial order between classes* occurs. A very common graphical representation of these classes and *ISA* relationship is the *class hierarchy* (see figure 4.3 and 4.4).

There is an important principle associated with the modelling of the *ISA* relationship. Basically any object is an instance of any ancestral class; this is the basis for what is called the principle of *subsitutability*. It is obvious that the extent of **NON-HUMAN_PRIMATES** is repeated (or absorbed) in the extent of **PRIMATES** and similarly in the extent of **MAMMALS**. Some literature tries to differentiate instances that are "direct" instances-of to those acquired through generalisation. An extent that includes all instances from any descendent class is called its *deep extent* (see figure 4.2). The following is a definition of the deep extent. The following says that the deep extent of a class is a union of itself with every deep extent of its subclass elements.

$$\prod Cn = \pi(Cn) \bigcup \{ id \mid id \in \pi(Cn_i) \ \& \ Cn_i \in C \ \& \ Cn_i \ ISA \ Cn \}$$

Other common terms include: a **PRIMATE** is a *specialisation* of a **MAMMAL** and that a **MAMMAL** is a *generalisation* of a **PRIMATE** [SMITH77]. A class that specialises another is called a *descendant* class and a class that generalises another is called an *ancestral* class

It is also important to study the meaning and restrictions on the instance-of relationships in the presence of an *ISA* hierarchy. In our biological system we have stated that the instances of **PRIMATES** and **CETACEANS** are *disjoint*. There also exists an *overlapping ISA* relationship that allows instances to participate in a number of sibling classes. In effect objects are now able to associate with a number of classes.

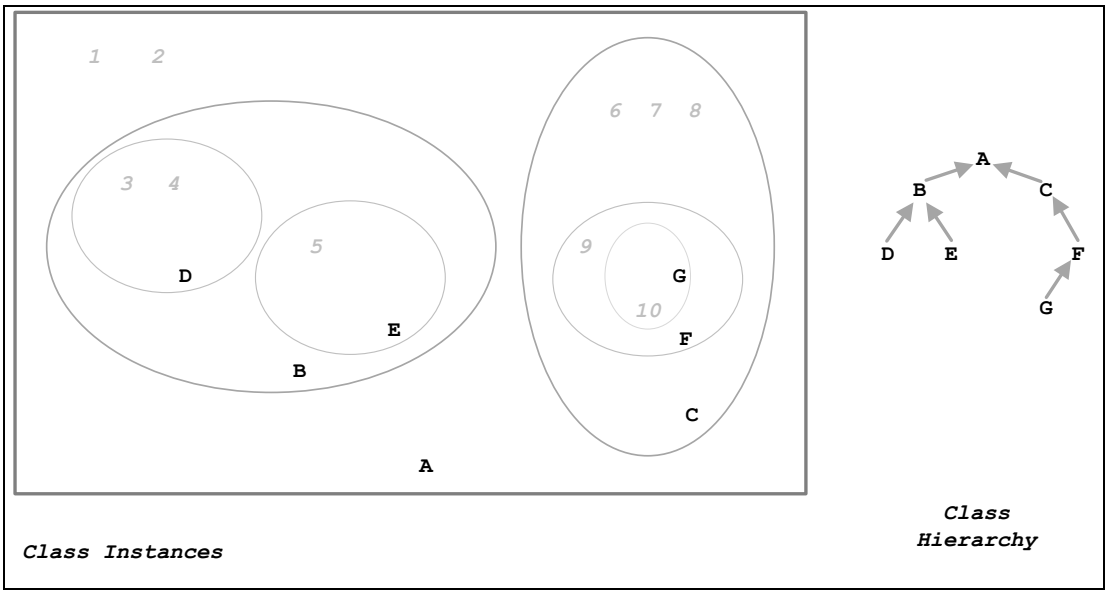


Figure 4.2 – *ISA* and instance of relationships. Classes are letters and instances are numeric. Class B *ISA* A, instances 3 & 4 are class D extent, and 3, 4 & 5 are class B's deep extent.

We have tacitly implied that given an animal instance we can classify it by finding the "right" leaf in the hierarchy. There are design issues as the following two cases expose. For example which classes cannot instantiate objects? These are called *abstract classes*. (In some systems this term denotes a class that provides an interface but no implementation). And do all instances of a class have to be an instance of some descendent class too? If affirmative we have a *total ISA* relationship otherwise it is a *partial ISA*– exactly like in an EERM sub-setting relationship.

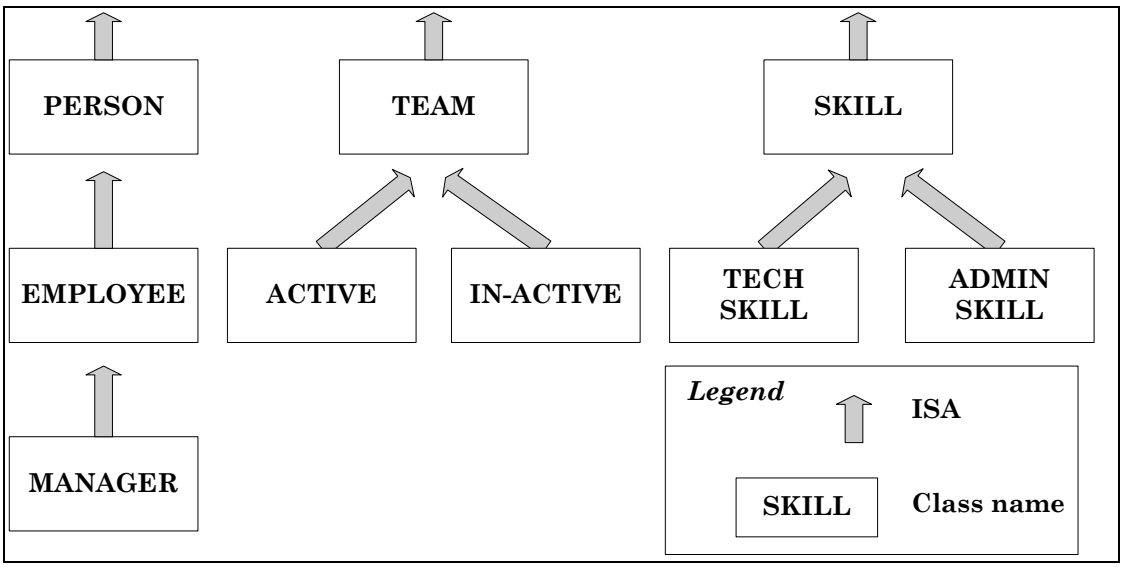


Figure 4.3 Classes and a Generalisation Relationship

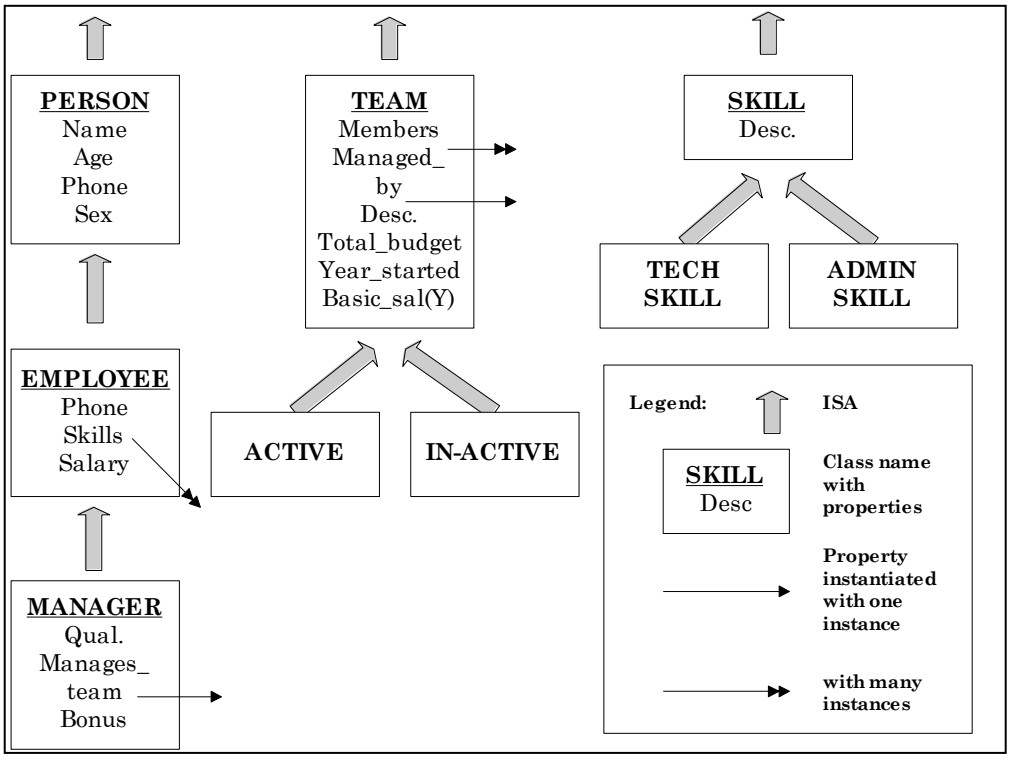


Figure 4.4: Class with properties in a hierarchy

Another interesting issue deals with the qualification of the *ISA* relationship. Let us assume we have a simple class called **PERSON** in which basic properties held include **NAME** and **ADDRESS**. The **MALE** and **FEMALE** classes have a property of interest that is **GENDER** and arbitrarily we rule that male instances have a "male" value whereas the female instances have a "female"; (note we can easily enforce this with integrity constraints – more

in chapter seven). Although we indicated that the instance-of relationship is of a semantic nature this is an example of a rule-based instantiation. We need to quickly state that if one swaps the gender's value that would imply an object migration from one class to the other – this is not typically an easy database operation and not surprisingly it is not implicitly available in most OODBMS.

4.1.2 Classes and Aggregation Abstraction

Although, as seen in the structured object explanation, an object composition allows hierarchic composition of values there are cases where an object is composed of a number of other objects through their identifiers. Also some of the "sub-parts" could also be an aggregated object.

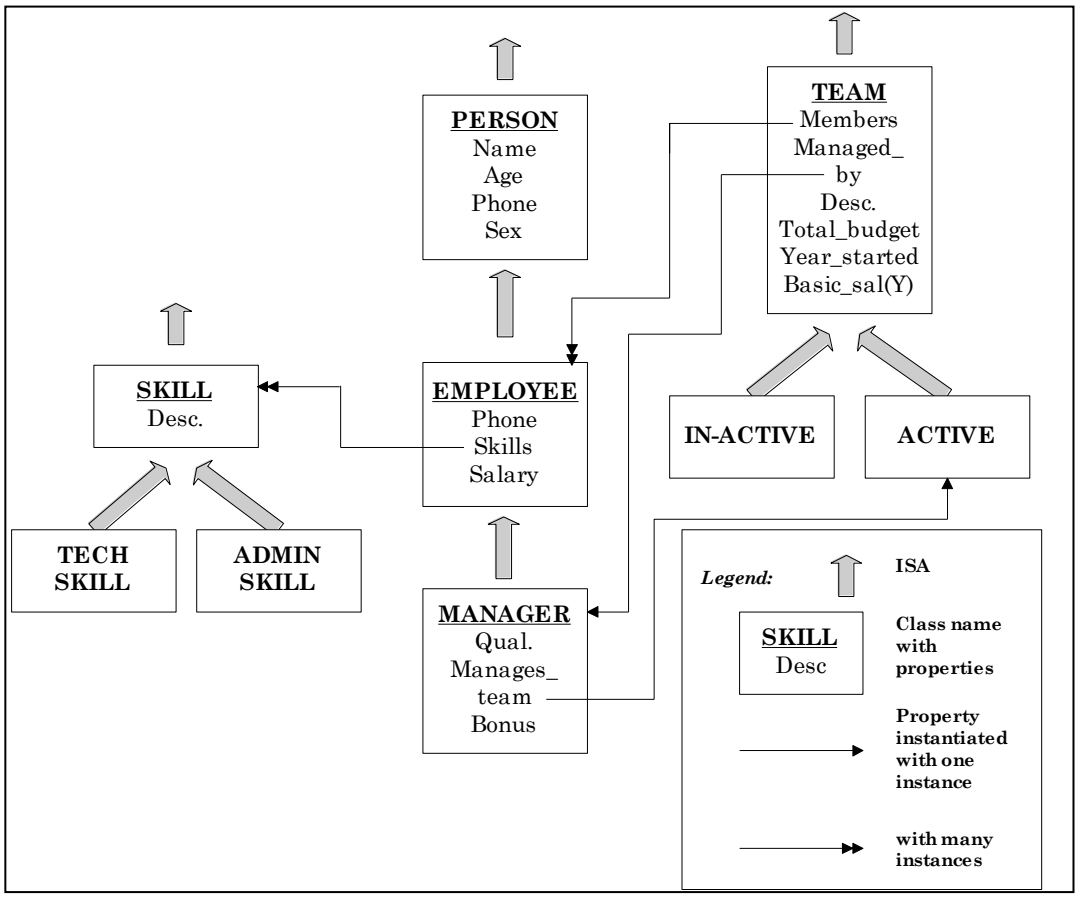


Figure 4.5: A Class Composition Hierarchy

Therefore in an aggregated object, represented in a class, each of its *aggregated relationship* takes the form of one of the following four types [KIMWO89G] through sharing and independence as discussed in chapter two.

A class definition example that includes an aggregated relationship follows.

```
Remark:  a fragment of a class definition with
Remark:   an Aggregation relationship in Orion's OODB prototype syntax
(make-class    VEHICLE
  :superclasses nil
  :attributes   ( ... ( engine_type :domain ENGINE
                        :composite true :exclusive false :dependent true)
                                  )
  :methods     ( ... ))
```

Another interesting and useful graph (i.e. in terms of data and query modelling) is the superimposition of the *ISA* and aspects of aggregation relationships in one graph, this is called the *Class Composition Hierarchy* [KIMWO90D] and an example is given in figure 4.5. Also part of chapter five that deals with path expressions has some relevance here. The structure of the graph follows that of the class hierarchy but what is added are another type of edges that denote a relationship between one class and another through composition and cardinality (i.e. set of single value).

4.1.3 Prebuilt Class Hierarchy

From the very earliest of object databases implementations, each had provided for a "pre-built" class hierarchy that had an array of class' implementations, with methods and attributes. These are useful "libraries" of code that provided software development a tangible boost to productivity through code reuse and differential development of the same library. It is instructive to attribute the in-built class hierarchy pragmatics, together with code development tools, to the Smalltalk [GOLDB88] project. Its design and tools are still valid today.

In figure 4.6 we have an example class hierarchy that includes both user-defined classes, for example **AMPHIBEAN** and its descendants, and prebuilt classes – for example **COLLECTION** and its descendants. The pre-built part of the class hierarchy is not usually drawn in an EERM.

4.1.4 Classes and their composition

We have presented classes as an abstraction for a composition of properties and as a partitioning agent for a domain's objects. From a data-modelling context these properties take a variety of forms. These forms include attributes, methods, relationships, class attributes and integrity constraints. Due to the database context these properties form part of a class' external interface and consequently the control of which properties an end user can read, update, delete or run falls in the non-trivial realm of database authorisation

modelling. An early articulation came from [CHAMB81] – which is not of any direct interest here.

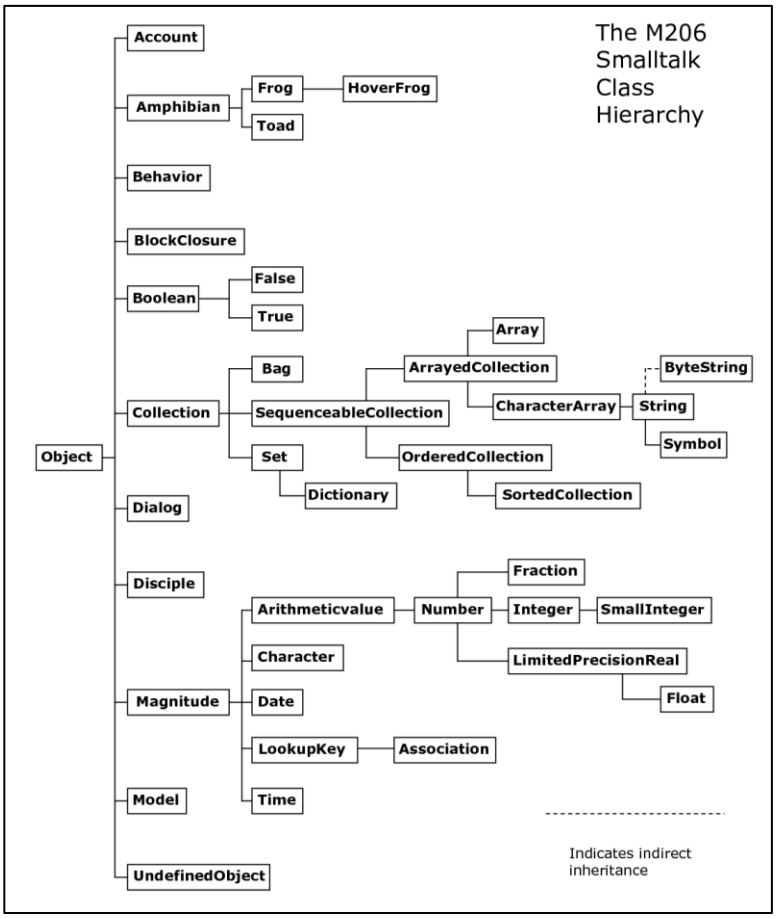


Figure 4.6 – Class hierarchy including pre-built classes. Source: McSweeney, J.; WWW.JMCSWEENEY.CO.UK/COMPUTING/M206/INDEX.PHP

A novel feature of the object-oriented paradigm is the use of methods as part of an object's composition. The more sophisticated class models also use an exception mechanism that invokes earmarked actions on an "error" event during a method's execution but does not return control to the method – i.e. non-resumable semantics. An early adopter of this exception model was C++ [STROU94]. Also, but unrelated to error handling, annotations on a method's blocking protocol (an excellent example is CORBA's *oneway* IDL method qualifier – see [BAKER97]) can also form part of a class definition.

An important part of a data model is the representation of relationships between objects. In a class structure this requires its own constructs that include details of the relationship that can adequately implement referential semantics and two-way traversal of a

relationship for n-way relationships (i.e. $n >= 2$). Also the "higher" form of relationships like *ISA* and aggregation has to be accommodated too.

Finally we have to include the provision of integrity constraints that are associated with the extent of a class and condition its behaviour. For example it is common requirement to have an object's attribute taking a unique value when compared to its class extent. Like an attribute, integrity constraints are applicable either at an object or at class-extent level.

4.1.5 Class Implementations and data types

Are classes and data types the same? In a class definition there is ample data-type details; for example in attributes, relationships, and methods. The main motivations for data types are expression correctness in terms of data usage and the possibilities to convert one data type to another if certain rules are respected. In fact it is advantageous to force a relationship between each class and a data type that describes it. We want to enforce:

$$Cn_i \; ISA \; Cn \rightarrow type(Cn) \subseteq type(Cn_i)$$

The interaction between classes and data types is based along two lines: each class has a data type and there could be many classes having the same data type, and some data types might not have a class associated with them. (A forthcoming section on Inheritance and Data Types shows the relationship between classes, data types and inheritance in more detail.)

Clearly we want to have classes and data types as two separate, yet related, artefacts.

4.1.6 Classification Critique

If classes and classifications are present in a data model then many aspects of database practices are assimilated; some are not. Consequently we have to acknowledge that choosing classes for classification modelling is a meta-design decision.

Before delving into class-based inheritance and data typing it is important to re-iterate the pros and cons of classification and classes because design choices need to be made on objective grounds.

One of the most important advantages is the partitioning of an application domain's objects. Not only can we classify but also, in general, querying (e.g. extracting) objects associated with a class extent is computationally easier than querying the whole database.

Through classification "higher" data-modelling constructs are possible. Also the class itself becomes a player in the data model as firstly it is a repository of behaviour, constraints and structural definitions and secondly it is able to hold aggregate data about the extent.

Although the notation, definition and implementation of a class is a straightforward matter the conceiving of a classification that meets the requirements of an application domain is another matter. Other than being "difficult" a classification is sometimes reduced into a match of political will and motives; for example the classification of wines. Also the design of a class presumes that the properties depicted in its definition are sufficient to describe and classify objects into their extents. What are we to do when the values of some object properties are unknown? What are we to do when a property, described in a class, is not applicable for an instance?

Re-classification of a class hierarchy is not a trivial matter as changes are to be on the classes involved and the instances found in affected extents and deep extents. At an object level moving it from one class to another is likewise a non-trivial task.

Another level of design difficulty with class hierarchies is when manipulating these as a whole. For example merging two class hierarchies is a major re-design event (for an early record of a class hierarchy re-design see [BERLI90] and [COOKW92]). Making any two class hierarchies co-operate over two diverse computational environments is also a design milestone requiring different and difficult and methodological procedures [BAKER97].

4.2 Inheritance

Inheritance is a mechanism for building objects by sharing properties of other artefacts; this is also attributed to Simula [DALHO66] too. This mechanism is almost synonymous with the paradigm and many positive features cited in favour of it are possible through inheritance. Yet it is obvious that inheritance means different things to different systems. For example in Artificial Intelligence inheritance it is used for knowledge representation, reasoning and inference. In semantic data modelling it enables the ordering of entities by an *ISA* relationship. In programming languages it helps in structuring and composing new data structures, and offering better opportunities for code sharing and code re-use – that is better productivity in development.

4.2.1 Inheritance Mechanism

Inheritance in object-oriented systems typically builds an object out of the properties specified locally for it and out of the properties found in other objects from which it is inheriting. Any property adopted is the one closest to the inheriting object, which can be defined locally or along the inheritance path; technically this selection is called *over-riding*. Also some systems prohibit redefinition of properties.

If a class system is in use then it is typical that class' *ISA* relationship determines the inheritance path and the encapsulation directive determines what is inheritable. The properties are, for example, attributes, relationships, and integrity constraints. Mixing the class concept and inheritance raises an issue; see next section.

The inherited properties are "modified", or *transformed*, to the local scope of the inheriting object during the runtime – a *late binding* operation. Specifically the transformation process needs to do at least two things. Firstly, it needs to check if an inherited property has been modified correctly (correctness will be defined later). And secondly, any inherited property with a "self" reference needs to have these references converted to the inheriting object (this is an important transformation as it distinguishes inheritance as a sharing mechanism rather than a copying mechanism).

4.2.2 What to inherit exactly?

Sub-classing down a class hierarchy implies that any subclass instance can be use when an ancestral class instance is required – it is called substitutability. Therefore any incremental modifications, done through sub-classing and inheritance, needs to conserve substitutability. Wegner [WEGNE89B] categorises incremental modification into four kinds of sharing each having its own set of constraints (or correctness) that influence what inheritance does. These are:

1) *Behavioural compatibility* where the inherited attributes and behaviour have type conformance and the same semantics – this is the strictest form of sharing. Behaviour is specifiable with a language that includes a signature and a semantic interpretation. The signature is a collection of methods and their data typing.

Behavioural inheritance compatibility has a number of themes: subset subtype, isomorphically embedded subtype, and object-oriented subtype. Nonetheless each

variant is bound to return the same object when properties (defined for both the parent and the child) are invoked with the same arguments on either of the parent's or the child's operations.

An example of the *subset subtype* is that the set of "integers from 0 to 100" is in relation to the set of "integers". A decisive problem to note here is that the restricted set is not closed for all its properties (the returned result of 50 plus 60 is out of the subset subtype range).

An example of the *isomorphically embedded subtype* is the set of "integers" is in relation to the set of "real numbers" with the addition and multiplication operations. A constraint that is applicable here is that if corresponding operations on corresponding arguments of the subtype are defined in the subtype whenever they are defined in the supertype and yield corresponding results. If the operations on the set "integers" are extended with division then we have lost this compatibility.

An example of the *object-oriented subtype* is an **EMPLOYEE** in relation to **PERSON**. In this case the subtype has to have its operations use the same domains for their arguments as those of their parent.

2) *Signature compatibility* is where the inherited properties may be extended horizontally by adding new attributes or extended vertically by constraining existing attributes. The inherited attributes and behaviour must be type conformant, but the inherited semantics is undefined and therefore may differ.

Here the signature is an approximation to the semantics. These subtype relationships are then checked at compile time for data type compatibility. Not all type compatible incremental modifications of the signatures are semantics preserving. For example, in a vertical signature incremental modification on **PERSON**, whose properties are **NAME** (domain string) and **AGE** (domain integers from 0 to 120), to **EMPLOYEE** (the **AGE** gets restricted to 16 to 65) some expression's type compatibility has to be validated at run time rather than at compile time. Furthermore setting the **AGE** of **PERSON** (who is also an **EMPLOYEE**) to 10 breaks our

signature constraint and also breaks the principle of substitutability associated with the *ISA* relationship.

3) *Name compatibility* (e.g. implementation models exemplified by Smalltalk) – where only the implementation is shared through the names of the properties. The inherited semantics can differ and type conformance of signatures (if present) is not a necessary condition.

4) *Cancellation* (e.g. Artificial Intelligence models) – where only some implementation is shared and some of the properties may be purged by the inheritor. Also this mechanism typically relaxes constraints down the hierarchy (in contrast to the previous three) – technically it makes the *ISA* relationship symmetrical. This is the weakest form of sharing. It is also to be noted that cancellation has an effect on our deductive systems in the sense that any arguments that could be derived for certain ancestor could become false in descendants that have cancel properties on which the original deduction would have been based. In this circumstance we have what is technically called *non-monotonic reasoning* (more in chapter six).

If we base our inheritance mechanism on a *strict compatibility* basis (behavioural, but not signature, object-oriented and cancellation) then the inheritance relationship is transitive and anti-symmetric. Consequently a partial order is established. This builds dependence between an object and any other object that inherits from it (already alluded to in the encapsulation section). A very important semantic constraint on any interpretation of inheritance is that the ordering does not contain any cycles (i.e. we do not want an object to inherit from itself). Our intention is to have the inheritance relation form a directed and acyclic graph. For the inheritance mechanism based on signature and object-oriented compatibility to yield a partial order we need to restrict its use.

After considering the inheritance mechanism it is worth delving into its reputed advantages. The more important of which are in conceptual modelling (which we have already presented as the *ISA* relationship) and in the possibility of re-use (and especially code re-use).

4.2.3 Incremental development through re-use

The reputed advantages of *code re-use* through inheritance are mainly three [MEYER96]. First is the possibility of building a system with a high degree of reliability – given that the sub-components are reliable. Second is the economic benefit from a reduction in the development phase of an application program. Third is the federated build-up of an object system by the incorporation of object definitions available from multiple developers.

Over and above sharing, an inheritance mechanism with late binding allows for an artefact development to be based on "*small and incremental*" changes rather than "*destructive development"* (or *overwriting*). In contrast to inheritance based incremental development is the "copy and paste" development mode. In this latter case any abstraction is brought down to the level of the copied items (e.g. lines of code) and, even more serious, this copied item isn't necessarily correct in its new context.

4.2.4 Single or Multiple Inheritance?

If the inheritance mechanism allows an object to inherit from many objects that are not related via an *ISA* relationship then we have *multiple inheritance,* otherwise we have *single inheritance*. Multiple inheritance offers more possibilities for incremental modification. Unfortunately this does force added difficulties both at conceptual and at technical levels.

At a conceptual level allowing for multiple inheritance shatters the meaning of conceptual specialisation especially since this mechanism doesn't imply any bias to a particular ancestor. The often-cited example is the **`ARRAYED_STACK`** class from Meyer's [MEYER96] use of multiple inheritance. This class inherits from both **`ARRAY`** and **`STACK`** with each class having clear behaviour and not related through an *ISA* relationship. Unfortunately the behaviour of the **`ARRAYED_STACK`** is not clear because it is easy to find examples where a property inherited from a parent class is not at all applicable to it. For example with the **`ARRAY`** class we associate the **`INDEX`** access which does not make much sense for any specialisation of the **`STACK`** class.

Inheriting properties through multiple inheritance potentially leads to *name collisions*. These collisions occur when the dynamic binding name look-up procedure finds more than one property that it should execute – see figure 4.7. In many cases the inheritance mechanism attempts to resolve these name clashes either by annotations written by the

designer (in Orion [KIMWO90D] the order of the class names in the inheritance list of a class definition determines a priority) or by using a superclass ordering mechanism (in CLOS the system linearises the directed acyclic graph and then chooses the property closest to the calling object [STEEL84]).

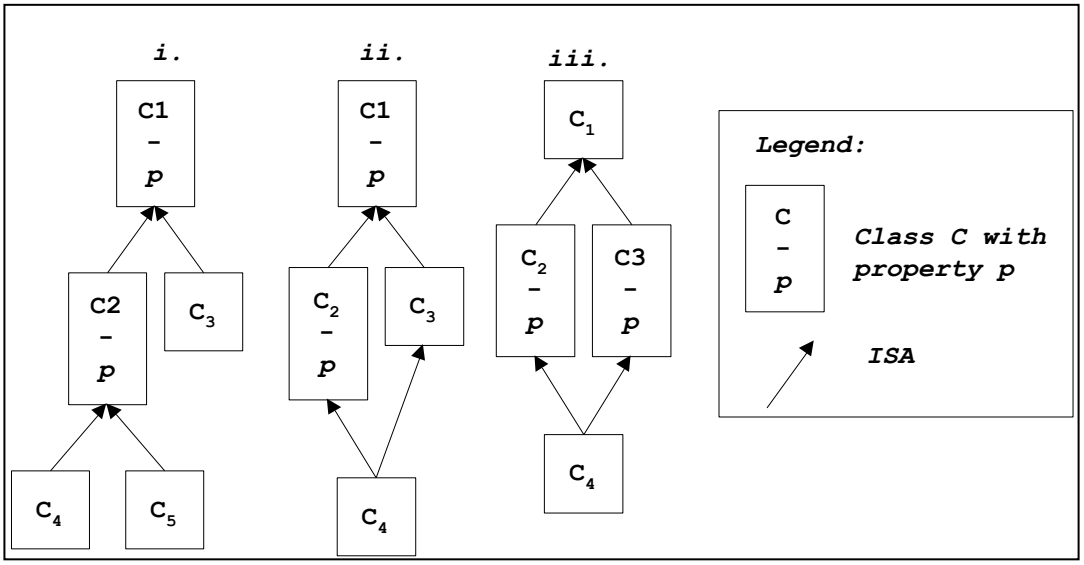


Figure 4.7: Different Occurrences of Name Collision in Multiple Inheritance

Can all name collisions be resolved? Indeed no, as can be shown in figure 4.7 and workings given in Simons [SIMON95]. In this situation a different property **P** is part of classes **C2** and **C3**. Also in C2 and C3 methods **R** and **S** depend on their respective P. In **C4** when it is to use method **R** it expects to use **C2**'s version of **P** while when **S** is used it now expects **P** from **C3**.

4.2.5 Inheritance in Conceptual Design and Implementation Design

We have seen how an inheritance mechanism can force a hierarchical relationship between classes. In this scenario the best inheritance mechanism compatibility will have to be the strict inheritance (i.e. behavioural) since through it we can safely make inferences on the relationships. But incremental modifications based on strict inheritance are not as flexible as Smalltalk's class based inheritance. Consequently it is common for object-oriented models to have non-strict inheritance even though the meaning in parts of the class hierarchy might be obscure or problematic.

To complicate matters it seems inheritance use, or modality, comes in two flavours. At a conceptual level we have the *ISA* relationship and at an implementation level where there

is an emphasis on re-use of properties down the hierarchy driven by convenience rather than compatibility (refer to Brachman [BRACH83] and America in [AMERI90B]). These inconsistencies are amplified when a designer tries to integrate a "shrink pack" class library with the application's own class requirements. Clearly a demarcation between the conceptual *ISA* relationships and developing classes through the inheritance of properties to facilitate incremental development (i.e. typically called sub-classing) must occur.

4.2.6 Other Points with Inheritance

Is there any other mechanism for combining classes with inheritance? One alternative is a system with a delegation-based inheritance where we have objects but no classes (Liebermann [LIEBE86]) and the mechanism is based on constructing objects (called prototypes) out of existing ones – this is the cloning process. In effect each object could have its own distinct set of properties. Delegation is common in programming environments but not in databases (at least those with a schema).

4.3 Data Types

When developing software artefacts our intention is to write these in a readable, reliable and efficient manner. Data types go a substantial way to help us achieve these intentions. A shallow description of a *data type* is a set of values and a number of functions and procedures that dictate how any element is manipulated. Typically how an actual data type implementation represents its state, functions and procedure is completely independent from its data type specification, so long as it does not break its own specifications. A data type specification might have a number of possible implementations.

Any operation on and expression involving a data item has to conform to the typing specifications of the attributes and functions of the data types involved. Before the operation or the expression is executed it has to be *type checked* against its typing specifications for any data type incompatibilities; if incompatibilities occur than there are *data type errors*. To check for this we need to be able to associate a data type with each expression and function call that is inductively built from the data specification and also from the data-type annotations given to intermediate results (e.g. the definition of a variable is followed by its data type). We say that every computational expression in a data typed environment has a *type expression* and type checking is the process of evaluating the

expression's data-type compatibility. If an expression's type check is successful then it is *type safe*. A *type safe program* must have all its expressions type safe.

C and C++ source code are not type safe programs when type casts and pointer arithmetic are present. On the other hand Lisp and Smalltalk have type safe expressions.

There are many reasons for a value, or an intermediate result, not to have a binding to a specific data type expression when an evaluation of the expression is required. Consequently one must attempt to garner data type details from partial data type specifications and data type bindings in the current expression. The process of using scant data type expressions from an expression and basic data type axioms to derive a full data type for an expression is called *data type inference*. For example in the following it is obvious, from the data type result of the SQL query, that the variable **NUMBER_PROJ** has to be an integer for whatever data type table **PROJECT** is.

```
SELECT count(*) INTO number_proj FROM project;
```

In any *data type system* we expect a minimum number of basic data types and these are typically the integer, the float, the character string, and the Boolean. Also, and as seen in the earlier chapters, a reference to an object is also useful for specifying sharing semantics. With each basic data type a number of applicable methods are defined. An important part of the data type system is the capability of constructing other data types, like the tuple *data type constructor*. The disjoint union allows for the construction of data type whose components are mutually exclusive (i.e. in an instance of the domain only one value is present). More sophisticated type constructors allow us to specify collections of values coming from the same data type. Specific example is the set of integers. The more expressive data type systems allow the creation of data type expressions based on polymorphic data types (a later section explains these). Some data type systems allow the specification of a data type to be given in terms of itself (i.e. *recursive data types*). An example of a recursive type is a list of integers.

4.3.1 Data Typing

If our data type system allows an artefact to take a value from one data type domain associated with it then our system is called *monomorphic*. The artefact could be a variable, an object, or an expression. Many programming languages are mostly monomorphic (e.g.

Pascal). If our data type system allows a variable to take a value from a range of data types then our type system is *polymorphic*. A good example is the ML [MILNE90] programming language data type system.

4.3.1.1 Static Data Type Checking

The two common modes of type checking are static and dynamic checking. Whatever type checking mode their core aim is common – prevent data type errors at run time.

In *static type checking* data-type errors are prevented by a number of measures. First, all objects, functions, procedures and variables have to be defined and annotated with typing information (sometimes redundantly through declarations). Given these data-type annotations then every expression's data type is understood though a "static" investigation of the environment (i.e. program's details) during program compilation. In this mode all data-type annotations are ironed out from the compiled version of the program; consequently the machine code generated is devoid of any type checking code. No code is compiled if any data type error is flagged.

Many literature sources assert that static checking forces the programmers to adopt disciplined code writing behaviour. Also the generated machine code is optimisable and efficient

Static type checking can be onerous when compiling a large source-code project (this is exasperated when the compiler cannot work out a clear order in the data type declarations [SCHUS11]). Static typing is a conservative typing technique in the sense that it prohibits a programmer from writing a general purpose program by leaving some of its coding with data type ambiguities that are resolvable at run-time – for example, a generically typed sorting algorithm. This implies that a program will not be compiled even when it could have been run without problems. Another problematic situation is the correctness of the design and implementation (in a complier) of a static type system when it is well known that these systems are complex and "immutable" once rolled out. A no holds barred and very one sided was written by Ousterhout [OUSTE98].

4.3.1.2 Dynamic Data Type Checking

Dynamically typed expressions imply that data-type errors are reported at run-time when they are not avoided. Consequently the compiled program needs to retain type details and

introduce type-checking code for the run-time data type checking of any un-resolved data-type expressions. Dynamic typing is useful as it allows a programming language to be used to write a generic type sorting. Another possibility with dynamic typing is the sharing and importing of persistent data from other environments [ABADI91].

There are issues with dynamic data typing; the most obvious is performance of its compiled code. Lately some authors are stating that 'just in time' compilation improves the dynamically type code performance. Nonetheless adequate performance is still found in more mature implementations like Lisp and Smalltalk. Another issue is lack of knowledge about a program during its coding. This manifests itself in debugging, documentation, and IDE environments (e.g. how to help code completion).

One of the first dynamically typed languages was Lisp, while Smalltalk is the earliest and best example from the object paradigm stable. Not many other programming languages offer wide support for dynamic type checking. The reasons being mainly two: first they make up for this through other mechanisms (e.g. ML relies on data type polymorphism), and second not many programming languages deal adequately with a persistent environment peculiarities (one can refer to the paper of Atkinson *et al.* [ATKIN87] on persistent programming).

Many programming languages have developed type systems with a balance of static and dynamic typing [MEIJE04]. Nonetheless static typing will always be required and conservative as identifying programs that produce run-time type errors is un-decidable. This follows straight from the halting problem.

4.3.2 Data Type Representation

Independent of when and how expressions are type checked is the requirement to represent the type systems. The two main streams are the algebraic approach (for example in the spirit of Guttag [GUTTA80]) and higher-order functional approach (for example as in Fun of [CARDE85]). In the *algebraic approach* we need to enumerate the data as sets of values called *ideals* and distinctly represent the functions (including their names and arguments' data type – called the *signatures*) that manipulate the data. Other than the function's signatures we need to express their semantics. The following example (from [DANFO88]) represents a simple algebra for a numbers modulo 2 system.

```
Remark:  modulo 2 algebraic system (from DANFO88)

Remark:  signatures

integer    : type;
zero, one   : function() result integer;
plus, times : function(integer, integer) result integer;

Remark:  semantics

integer     == {0,1};
zero()      == 0;
one()       == 1;
plus(x,y)   == if (x==0) then y; else if (y==0) then x; else 0.
times(x,y)  == if (x==0 or y==0) then 0; else 1.
```

Although the previous is a common *Abstract Data Type* (ADT) representation and is an effective tool for abstraction it suffers from not having a concept of state. These systems work by requiring the programmer to provide for a state (e.g. an integer variable) and then invoke the operations from the above *modulo 2* ADT with an integer as an argument and then wait for the result of the operation. The separation of the state from its associated ADT is sometimes scorned. Parnas [PARNA72] does advocate for state and ADT collective representation.

4.3.3 Data Type Inference

We have previously stated that each expression's data type is built inductively from its components' data-type annotations. What if some data-type annotations are missing? It is likely a type checker cannot check such an expression. Nonetheless the level of type safeness can be increased if type checking invokes a process of deducing the data type of expressions from the type declarations found in the program, the context of the expression, and the axioms of the type system. In such circumstances and in the right programming frameworks *data type inference* is an effective development tool. During the bottom-up type checking of an expression if a variable has no associated data type then a *data type variable* is introduced. Consequently for a successful data type inference process the system has to come with an instantiation for each data type variable that does not contradict other data typing constraints either given or implied.

For example in the following SQL statement if our knowledge of the **SYSDATE** function includes the return type then it determines the variable's, i.e. **TODAY**, data type. Therefore we don't really need to annotate the variable as a date.

```
SELECT sysdate INTO today FROM dual;
```

If we know precious little about the **SYSDATE** function then we cannot infer **TODAY**'s data type.

Again we point at ML [MILNE90] and emphasise its flexible type system whose type checking has a type inference system component based on first-order logic assertions and deduction. A number of programming languages are introducing type inference in their respective revisions.

We have implied, rather naively, that type checking and type inferences are embedded in an effective procedure. In fact some extensions to a basic type system can make both checking and inferring un-decidable. Such extensions are required for aspects of the object-oriented paradigm – some of Palsberg questions still remain open [PALSB96].

4.3.4 Data Type Theory

At this point an introduction to the semantics of a type theory that caters for abstraction and universal polymorphism in the object-oriented paradigm is opportune.

V, the *set of all values*, is Scott's [SCOTT76] universe of all values built from basic domains, records, disjoint union and function spaces. This set's elements are in partial order. A type can then be described by a set of elements from *V*. Not all subsets of *V* are acceptable or useful types but those that are we call *ideals*. Some of these ideals are the familiar data types found in programming languages. *Set inclusion* is proposed as an ordering relation over the ideals thus forming a lattice structure. Therefore the *type hierarchy* generated is graphically analogous to class hierarchy presented earlier. The apex and end nodes of the lattice are *V* and the set with the least element of *V* (the empty set) respectively.

The notion of a value having a type is interpreted as membership in a corresponding ideal. The selection of basic domains and type constructors influence important characteristics of a programming language's data-type sub-system. Therefore a monomorphic type system does not allow a value to be member of more than one type (i.e. ideal). On the other hand a type system that allows a value to be a member of different types is a polymorphic type system.

4.3.5 Object-Oriented Data Typing

Two evident data-type characteristics with the object-oriented paradigm are subtyping and polymorphism; actually these two are related. The overlying principle is that of *data type substitutability*; can an object, of a different and perhaps related type, be used instead of the expected one without a type error being flagged? There are many shades of substitutability; Liskov in [LISKO87] & [LISKO99] claims that substitutability is meant to maintain semantic properties rather than a weaker subtyping constraint through a properties' signature.

With our notation of sets as types we can map the idea of subsets to sub-types. The expression that a *data type B is a subtype of data type A* ($B \subseteq A$) implies the data type-set of *B* is in a sub-set relation to the data-type set *A*. This is a very elementary notion but applicable for explaining sub-range types and some subtyping manifestations.

Yet another example extensively used by Cardelli [CARDE84] is the type inclusion between record structures with functions; these represent attributes and methods. Incidentally this representation has since been used by many others to study "inheritance" between records.

A record type **A** is a sub-type of a record type **B** if **A** has all the attributes of **B**, and possibly more, and the common attributes are in a sub-type relation ($\mathbf{A_i}$ denote attribute names, and $\mathbf{T_i}$ and $\mathbf{U_i}$ are data types):

$$\mathbf{IF}\ \{A_1:T_1, \ldots, A_N:T_N, \ldots, A_M:T_M\} \subseteq \{A_1:U_1, \ldots, A_N:U_N\} \leftrightarrow T_I \subseteq U_I\ \mathbf{FOR}\ I = 1\ \mathbf{..}\ N$$
$$\mathbf{THEN}\ A \subseteq B$$

In terms of substitutability based on attribute signature it is evident that A can take the place of B. Every type is a subtype of itself; $A \subseteq A$.

Subtyping for a function is:

$$\mathbf{IF}\ S' \subseteq S\ \mathbf{AND}\ T \subseteq T'$$
$$\mathbf{THEN}\ F{:}S \rightarrow T \subseteq F{:}S' \rightarrow T'$$

Two important characteristics to note are the *variance* of the domain (the sub-typing between **S'** and **S**) and the *contra-variance* of the range (the sub-typing between **T** and **T'**).

One of the basic characteristics of object-oriented systems is data polymorphism. Data type polymorphism comes in many disguises and one of the first researchers to classify data polymorphism was Strachey [STRAC67]. Strachey divided polymorphism into two sections with the first called universal polymorphism (actually Strachey called this parametric) and

the second called *ad hoc* polymorphism. Later Cardelli and Wegner in [CARDE85] refined this classification by adding polymorphism with subtyping.

4.3.5.1 Universal polymorphism

In *universal polymorphism* we have a method whose code is applicable over a collection of data types that have some common characteristics. This category is divided in two overlapping sections basically parametric and inclusion polymorphism. *In parametric polymorphism* a function definition is adorned with a type variable that has to be instantiated with a data type during a data-type checking of an expression involving this function. In this example the swap function arguments are assigned a type **FLOAT** when being applied over variables **HIGH** and **LOW**.

```
Remark:  parametric polymorphism example
Remark:  using c++ templates

template <typename DT>
void swap(DT& left, DT& right)
{  DT temp = left;
   left    = right;
   right   = temp;
}

Remark:  in use i.e. declaration and invocation

float high, low;

swap(high, low);
```

Universal quantification implements data type polymorphism at a second-order level. During data-type checking and inferring the procedure for binding of parameters to actual values needs to be done at two levels of abstraction. The first is for data types and the second is for values.

In the case of a recursive data type it is understood that parametric types are not expanded by a macro replacement scheme (i.e. that leads to an infinite expansion). There exist problems when trying to establish when two parametric recursive type definitions are of the same type. Typically one solves these problems by placing syntactic restrictions on the parametric type definition.

4.3.5.2 Existentially quantified polymorphism

Type expressions that are *existentially quantified* assume the availability of a type that satisfies them. There may be many types that satisfy an existentially typed expression and therefore interesting solutions to these are those expressions that are somewhat restricted

by other explicit data-type annotations in the expression being solved. Consequently, the existential quantification alters the whole body of a data type. Existential type expressions are useful and especially if they contain enough structure. Most ADTs are considered as existentially quantified [MITCH88]. These types manage to hide some structure of an object instance but still show enough structure to allow manipulation of objects through the operations these data types themselves provide.

```
Remark:  existential quantified types example

Remark:  declaration of a module with an attribute and a
         function with Boolean as its range

EQDType  = ∃ dt { attr:dt; f(dt → Boolean); }

Remark:  a possible implementation of EQDType follows

stringDT = { attr:string; f(string → Boolean); }
```

Some data-type expressions can use both universal and existential quantification for pronounced flexibility. Cardelli and Wegner [CARDE85] give an example of a parameterised stack whose data type expression includes existentially and universally quantified type variables.

4.3.5.3 Universal Bounded polymorphism

Bounded quantification in a data-type expression allows the provision of replacing data type variables with explicit sub-types of the parameterised data types. Inheritance is modelled by explicit parametric specialisation of types to the sub-type for which the semantics will be evaluated. The earlier presentation of sub-types based on subsets has opened a variety of interpretations dependent on the type constructor used in the data-type expression

The "for all" capability of universally quantified type variables in a generic function might actually be more than we would want to bargain for. In some cases we want to restrict the quantification over a restricted set of subsets (e.g. to those that are sub-types of a given data type). *Bounded universal quantification* helps us express these data type expressions. Not only is this a useful specification technique but it makes our type system even more expressive. When we apply a function on a bounded universal quantification data type expression the "extra" data type characteristics are not stripped off.

```
Remark:  bounded quantification in java
Remark:  source Angelika Langar's Java Generics FAQ at
Remark:  www.angelikalanger.com

class ClassI {}

class ClassJ < DT extends ClassI >    # datatype DT is ClassI or any
descendent
{     public ClassJ identity( DT whatever )
      {    return whatever; }
}
```

Similarly existentially quantified type parameters in expressions can be bounded with the sub-type relations. In some systems with bounded quantification type checking has been shown to be un-decidable [PIERC92].

4.3.5.4 F-Bounded polymorphism

Unfortunately there is a lurking problem within bounded quantification over type expressions with recursive types. In Canning *et al.* [CANNI89] this problem together with its solution were investigated. Namely the authors showed how type-checking perfectly common object-oriented expressions couldn't be typed with the data-type theory presented in Cardelli and Wegner [CARDE85]. The expressiveness of bounded quantification drastically diminishes in the presence of recursive data types because some methods are not "inherited" over the data-type quantification because of subtype violation. The solution presented by Canning *et al.* entails the introduction of another sub-type relationship in the data type expression so as to relate the necessary methods through the sub-type relationship. Their method is called *F-bounded polymorphism* and an example follows.

```
Remark:  f bounded quantification as in Canning et al. [CANNI89]

integer {…, lte:integer->Boolean, …}   # note integer is recursive
porder  { lte:porder->Boolean }        #      porder is recursive too

min: ∀dt dt ⊆ porder  (dt, dt) -> dt;  # is a method for porder

We claim:
if ( integer ⊆ porder )
then { …, lte:integer->Boolean, … } ⊆ { lte:porder->Boolean }

In which case the subtype relationship between functions gives:
i)    Range:  Boolean ⊆ Boolean        # No problem

ii)   Domain: porder ⊆ integer         # By argument contravariance
                                       # Problem contradiction

Therefore either two types are the same or original claim is wrong.

F-Bound solution starts to redefine porder and create integer
fb-porder[dt]  { lte:dt->Boolean }        # porder is not recursive

fb-porder[integer]  { lte:integer->Boolean }
```

```
Hence:
integer ⊆ fb-porder[integer]

and the new function min works as expected
min: ∀dt dt ⊆ fb-porder[dt] (dt, dt) -> dt
```

Java's later versions [GOSLI05] compilers support F-bounded quantification.

```
Remark:  F-bounded quantification in java
Remark:  source Angelika Langar's Java Generics FAQ at
Remark:  www.angelikalanger.com

class ClassI< DT > {}

class ClassJ <  DT extends ClassI< DT >  >
{     public ClassJ identity( DT whatever )
      {    return whatever; }
}
```

Serendipitously F-bounded quantification indicates another important observation about the relationship between inheritance and sub-typing, that is two types that satisfy the F-bound might not be in an inheritance relationship [CANNI89].

In *inclusion polymorphism* a function's data type variable can be satisfied by a data type and other data types somehow related to the former characteristics. The introduction of inclusion polymorphism addresses some of the pragmatic notions of object-orientation, namely inheritance and sub-typing.

```
Remark:  inclusion polymorphism using c++ classes in humans.h

class person
{   public:
    virtual void title() = 0;
};

class employee : public person
{   public:
    void title()
    {  std::cout << "employee! \n"; }
};

class consultant : public person
{   public:
    void title()
    {  std::cout << "consultant! \n"; }
};

Remark:  in sillyapp.cpp

#include <iostream>
#include "humans.h"

void print_title(person *employee)
{    employee->title();
}

int main()
{    empl employee;
     cons consultant:
     print_title(&empl);                # prints employee!
     print_title(&cons);                # prints consultant!
}
```

4.3.5.5 *Ad hoc* polymorphism

In *ad hoc polymorphism* a function works on a number of data types but the function's code is typically re-written for each data type. The data types on which an *ad hoc* polymorphic function operates don't in general need to have any common characteristics. The behavioural meaning of an *ad hoc* polymorphic function is usually different for each data type instantiation it can be invoked on. This "apparent" polymorphism manifests itself into two main disguises (that aren't entirely distinct).

The first form is called *overloading*; and this has been previously described. The second example is called *coercion* and this involves inserting clear procedures of how to force a data value of a certain domain into a value of another domain that is being expected by the function. Cardelli and Wegner warn the readers with the following "ALGOL68 is well known for its baroque coercion scheme" [CARDE85]. In C and C++ [STROU92] explicit type coercion, for example, is done with **`(INT)`** and **`STATIC_CAST`**.

4.3.6 Advantages of data types

What are the advantages of having a well embedded and a well-grounded type system in the object-oriented paradigm? Danforth and Tomlinson in [DANFO88] iterate the following:

1) Types provide a uniform framework with which to understand the objects.
2) Within declarative languages (where denotations are the means of guiding computation and few explicit type expressions are given) it is advantageous to have a type theory capable of explaining and representing the meanings of expressions.
3) A high-level typing theory aids the construction of efficient, useful and understandable software.
4) Inter-object communication trends can be invaluable in achieving locality of access in parallel systems and can also be useful in load balancing. Typing systems could help compiler optimisation.
5) Type secure modules could be achieved by dynamic or static typing checking (see later definitions).

What are the disadvantages of a type system? Firstly, while all programming languages have the same power, as each is Turing complete, each language compilable source is

affected by its type system. Secondly, the opportunities presented by object-oriented polymorphism, their resolution and the eventual pruning took its time to develop, be accepted, and be implemented.

4.3.7 Inheritance, Sub Typing and Sub Classing are not the same thing

We have just argued that sub-typing is one of the mechanisms found in the object-oriented paradigm that realises data-type polymorphism. We have also seen how a certain interpretation of sub-typing can partially order data types into a hierarchy. Also the data type system aims at making type checking of expressions reliable and efficient.

When we considered classes, as a composition of properties, with a semantic relationship (i.e. instance-of) with its instantiations a basic hint given was that each class was implementing a data type. After our exposition of data types it is obvious that each class has an association with a data type. Classes too have been organised in a hierarchy through a class *ISA* relationship. We must reiterate that the underlying motivation for this hierarchy is code re-use. It was made apparent that specialisation (i.e. a conceptual design feature) and pragmatic re-use machinations are not entirely compatible.

One of the first literature sources to start affirming this disparity was Brachman in [BRACH83]. As for prototypical research projects the first that practised what it preached, namely inheritance and "sub-typing" are different, was the POOL language ([AMERI87] and [AMERI90B]). At a later time pragmatic (e.g. Porter in [PORTE92]) and theoretical work on F-bounded polymorphism showed that sub-typing and inheritance were not always in harmony.

If we had to survey some of the current object-oriented programming systems it transpires that these systems adopted a single hierarchy system (either type or class based – and these are not exactly in line with our definitions here) and inheritance is controlled by sub-typing rules. The current stable of object-oriented data type systems offer a strong and effective facilities; as exemplified in Self (Smith and Ungar [SMITH95]), Java (Gosling [GOSLI05]), Scala (Odersky *et al.* [ODERS10]), C# (Hejlsberg *et al.* [HEJLS10]), and Links (Cooper *et al.* [COOPE06]).

Given that we accept the need for the two hierarchies what would be the major hurdles. Basically the complexity of the system becomes harder still. This hurdle was significant for the paradigm at an early start [BERLI90] but is nowadays attenuated with much better language design and education. Also the fact we have separated the two semantic structures does not mean that their developers and users would not "abuse" of these and there-by constricting the pros of the dual system. On the other hand the best advantage of de-coupling the two hierarchies (but keeping a class associated with a data type) is the resolution of many sticky problems when coercing classes onto types or types onto classes.

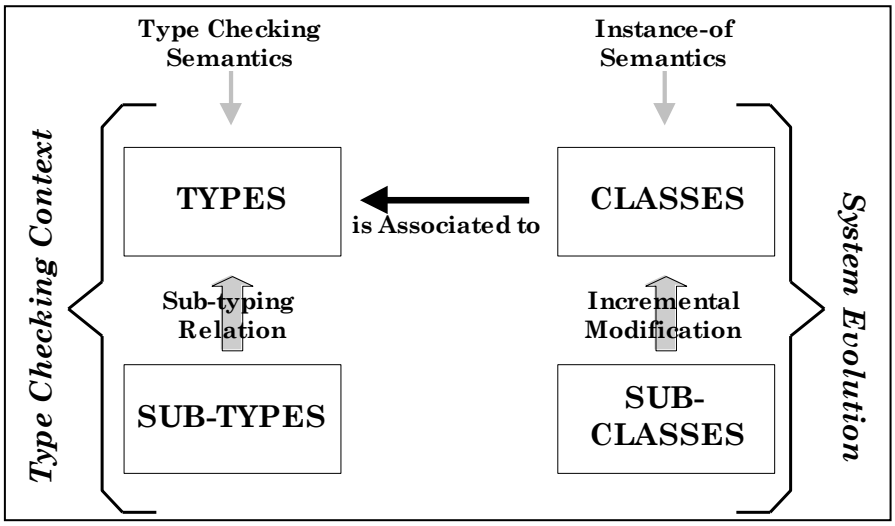


Figure 4.8: De-coupling Sub-classing and Sub-typing

4.4 Summary

If one adopts the class theme in an object-oriented database then a number of data modelling possibilities are possible. These include classification of the object collection, composition of a class' properties, specialisation and generalisation through the *ISA* relationship between classes; and build-up of involved structures through aggregation. Classes are also used to implement data types through, for example, the declaration of their properties' data type signatures.

The inheritance mechanism is an effective technique for sharing properties to achieve artefact re-use. For example, it is used to re-use code through incremental code build-up.

Data-type systems are concerned with the correct application of data to expressions. Data type checking models include static and dynamic systems. In many object-oriented systems dynamic typing is employed and it is coupled with data-type inference to work out the type

correctness of expressions at runtime. Also subtyping, a specific category of data-type polymorphism, is used to improve data typing specification and implementation. Furthermore the paradigm offers a variety of other data-type polymorphisms and given that known type transformations are used, like F-bounded polymorphism, allows the designer to build more robust data-type regimes to support the class hierarchy implementation.

After this last two-chapter into object-oriented themes and variations the next step is to complete the survey with an emphasis on databases and the ODMG standard.

Chapter 5

Object-Oriented Paradigm: OODB and the ODMG Standard

5 – Object-Oriented Paradigm – OODB and the ODMG Standard

The first major research efforts in object databases started in the late-eighties, these include *Postgres* [STONE90A], *Orion* [KIMWO90C], *Iris* (*OpenDB*) [KENTW91A] and O_2 [DEUXO91, ATKIN92], and each had their own data model. It was natural that the database research community and the product's suppliers wanted to harmonise the object database diversity. The first major effort toward this harmony came from Atkinson's *et al.* [ATKIN90] and in it the authors enumerate important themes and features.

There are advantages when modelling through object orientation for example, objects include details of state and behaviour and also the state includes more details of its relationship with other objects.

In this chapter we build a definition of a class-based, object-oriented database and it has a basis from the previous chapters (i.e. three and four) on object-oriented themes. Also a subsection presents collective knowledge of path expressions. The final section is a detailed study, exposition, and critical overview of the ODMG data and query model. (Further details are found in Appendix – ODMG Data Dictionary). Also in the final section an OODBMS, EyeDB, which has a good coverage of the ODMG standard, is presented. EyeDB also offers OODBMS features not included in the ODMG standard.

In coming chapters the EyeDB is used as a test OODBMS to translate an EERM diagram into a schema specification that has both structures and constraints.

5.1 OODB schema and instance

In this chapter we tie in classes, types and objects with object databases through schemas. An object database schema is split into two parts: one structural and the other behavioural. With the structural schema we define each object's collection that satisfies its constraints including the *ISA* relationship.

The formalism here, and in earlier section (i.e. 3.2-3.4, 4.1.1, and 4.1.5), follows Kannelkalis [KANNE92], Abiteboul [ABITE93A], and Straube [STRAU90A]. Well known textbooks include Delobel *et al.* [DELOB95], Lausen *et al.* [LAUSE97], and Abiteboaul *et al.* [ABITE95].

We can assume that we have a class that is arbitrary set to be the ancestor of all classes – variously called 'root class' or 'object'. Correspondingly we also assume the presence of a type ANY.

As for the behavioural schema we indicate the implementation and name resolution requirements.

The structural schema, is SS = ($\mathcal{C}$, $\mathcal{ISA}$, $\mathcal{Type}$)

The structural schema is a triplet of classes, $\mathcal{ISA}$ relationship between classes, and a mapping called $\mathcal{Type}$ from a class to its data type definition. $\mathcal{Type}$ is based on *type* depicted earlier, e.g. figure 4.8, but it reflects any sub-typing specified and the fact that a type is assigned to each class.

$$\forall(Cn, Cn_i), \qquad Cn_i \; ISA \; Cn,$$

$$Type(Cn) == type(Cn) \cup type(Cn_1) \cup \cdots \cup type(Cn_n)$$

and

$$\forall(Cn, Cn_i), \qquad Cn_i \; ISA \; Cn \rightarrow type(Cn) \subseteq type(Cn_i)$$

An instance, *I*, of a structural schema just given, is a pair of extent mappings and object states (i.e. *obj*).

$$I(\text{SS}) = (\pi, \; obj)$$

Where π must ensure:

$$o \in \; \pi(Cn) \; \& \; Cn \in C \; \rightarrow V(o) \in \; \|Type(Cn)\|_{\pi}$$

Also note that $\mathcal{V}$(o) is not v(o) in that $\mathcal{V}$(o) includes inherited attributes, for class Cn, and assigned values to these.

In figure 5.1 we can appreciate schematically the mappings between classes, types and instances.

$$\begin{array}{lll} C & \overset{Type}{\longrightarrow} & T \\ \Pi\downarrow & & \downarrow[\![T]\!]_{\pi} \\ O & \overset{\mathcal{V}}{\longrightarrow} & obj \end{array}$$

Fig. 5.1 – Mapping of Classes, Types, and Objects with Inheritance

The behavioural schema is defined through a set of methods, each of which is assigned to a class. To describe a method its name (i.e $\mathcal{Mn}$), and type of arguments and return data type are given. The schema is a triplet of Classes, the $\mathcal{ISA}$ between classes, and the set of method signatures, $\mathcal{S}$. The return data type can be a set of or scalar.

The behavioural schema is BS = ($\mathcal{C}$, $\mathcal{ISA}$, $\mathcal{S}$).

The signature set, therefore there are no duplicates, is made up of entries that have the following form (slightly different from previous notation as now we have a class as one of the arguments):

$$Mn: Cn \times T_1 \times \ldots \times T_n \rightarrow RT$$

As described in the earlier section data types (i.e. 4.3), method signatures are expected to uphold the variance and contravariance of the argument and return type data type. Consequently a method, say *mth*, is executed if:

$$mth(o, \mathrm{o}_1, \ldots, \mathrm{o}_\mathrm{n})$$

$$o \in \prod Cn, \quad o_i \in \|T_i\|_\pi$$

Instances of behaviour schema are expressed as a triplet of class extent, method implementation, and method resolution.

$$I(\mathrm{BS}) = (\pi, mthImplement, mthResolution)$$

For method implementation we have a partial function:

$$\forall s \in S, S = (Mn: Cn \times T_1 \times \ldots \times T_n \rightarrow RT) \rightarrow$$

$$mthImplementation: [\![Cn]\!]_\pi \times [\![T_1]\!]_\pi \times \ldots \times [\![T_n]\!]_\pi \rightarrow [\![T]\!]_\pi$$

For method resolution we have another partial function from methods and classes to a signature:

$$\forall s \in S, S = (Mn: Cn \times T_1 \times \ldots \times T_n \rightarrow RT)$$

And if:

$$Cn_i \; ISA \; Cn_j$$

and no $S = (Mn: Cn_i \times T_1 \times \ldots \times T_n \rightarrow RT)$

is found then any of the following is adequate to resolve the method call:

$$S = \left(Mn: Cn_j \times T_1 \times \ldots \times T_n \rightarrow RT\right)$$

5.2 Path Expressions

In earlier sections we pointed out that direct- and value based access to an object is possible in the object-oriented paradigm. Given that an object structure is rich and includes implicit inter object relationships, could we use this structure to access embedded objects? Similarly we are also interested to know which object is related to which other objects and under which circumstances. Ideally when we are holding an object's reference, e.g. a team instance called **TEAM_TEN**, then through its **MANAGED_BY** property we can then access its office **PHONE** (see figure 4.5). A common notation uses the dot to specify the following object chasing:

```
Remark: example i)
team_ten.managed_by.phone          # e.g. return value is 9922 4455

Remark: example ii)
team_ten.managed_by                # e.g. return value is #id2233

Remark: example iii)
team_ten.managed_by.bonus          # e.g. return value is 20000
```

This notation of denoting objects is called *path expression* and is accredited to Zaniolo's work on GEM [ZANIO83]. The leftmost artefact, i.e. **TEAM_TEN**, is called the *selector* and it is usually either a class or an object identifier (e.g. even a named object as in this case). The rightmost artefact is called the *return value* of the path expression. A path expression evaluation can return null, one or many values or identifiers. The intermediate artefacts are attributes that follow a path from the selector to the return value along its class composition hierarchy (see figure 4.5). A path expression is really an *implicit join* (e.g. implemented through part-of relationship or a complete referential constraint) and qualified by the instance-of and *ISA* relationships. For the third example we start from class **TEAM** instance named **TEAM_TEN**, traverse through **MANAGED_BY** to class **MANAGER,** and return the Manager's **BONUS** value.

Path expressions are extensively used in object-oriented databases and are a fundamental requisite of query modelling. Important references include the following: Orion project [KIMWO90D] and later additions with XSQL [KIFER92A], Bertino's [BERTI89], and Frohn's [FROHN94]). Path expressions are the basis of the XPath [BENED08] language and many

semi-structured data models and implementations. Another interesting addition to UML is its Object Constraint Language; visible contributors to this effort are Warner and Kleppe [WARNE03].

5.2.1 Path Expressions in more Detail

An object's properties include attributes, relationships and methods. If we collapse attributes and relationships into methods as methods that take no parameters then we have a convenient notation (i.e. methods with typed parameters and typed result) to express the capabilities of path expressions. We have to be particular on the type constructor of a method's result type. Namely whether it is a single object (value) in which case we call the method result data type constructor scalar or whether the method returns a set of objects (values) in which its result data type is a set.

5.2.1.1 Scalar Path Expressions

The above examples (i, ii & iii) are of the scalar path expression type. In the following example two path expressions are checking whether the team's manager phone number is the same as that of object being referenced by **THE_BOSS**:

```
team_ten.managed_by.phone == the_boss.phone     Remark: returns a Boolean
```

Scalar path expressions can also make use of methods. For example assume that the **EMPLOYEE** class has a method called **BASIC_SALARY** (a partial function) that returns a year's salary given that particular year (an integer) is passed as a parameter.

```
the_boss.basic_salary(2013)                     Remark: returns 25000
```

5.2.1.2 Set Path Expressions

A simple example of a set path expression follows (based on figure 4.5); this expression denotes a number of skill objects (instances from the class **SKILL**) that a particular **EMPLOYEE** (whose identifier is **EMPLOYEE_15**) holds. It is worth noting that expression evaluation includes instances from the deep extent of class **SKILL** (i.e. classes **ADMIN** and **TECH** skills). Therefore the data type of the return expression is heterogeneous (albeit associated through *ISA* relationship and possibly through data type compatibility).

```
employee_15.skills                  Remark: returns 'coding' and 'testing'
```

Set path expressions are more varied than scalar path expressions. For example a set path can be compared with another set path or compared with a scalar type. Here follows some examples.

Is **EMPLOYEE_15** part of **TEAM_TEN**. The "includes" operator's structure is "set includes item" and its meaning is true if "item" is found in "set".

```
team_ten.members include employee_15     Remark: returns Boolean
```

Are all of **TEAM_TEN** employees with a **SALARY** greater than 10000. The "all>" operator's structure is "set all> item" and its semantics implies that all of the set's elements are greater in value than item (for example the expression "{3,4,5} all> 2" is true). This is an example of universal quantification.

```
team_ten.members.salary all> 10000       Remark: returns Boolean
```

Is there an employee on **TEAM_TEN** that has the same **PHONE** number as their manager. The "any=" operator's structure is "set any= item" and its meaning is true if there is at least one item that is equal to an element found in the set. This is an example of existential quantification.

```
team_ten.members.phone any= team_ten.managed_by.phone Rem.: ret. Boolean
```

In a set path expression methods have a role too; for example if we are interested in collecting the salaries for the year **2013** of **EMPLOYEES** associated with **TEAM_TEN**.

```
team_ten.members.basic_salary(2013)         Remark: returns Boolean
```

As a method's parameters could be any object expression (so long as the data type is compatible) then even the year could be substituted by a path expression (in this case a scalar one that returns an integer).

```
team_ten.members.basic_salary(team_ten.year_started) Rem.: ret. Boolean
```

5.2.2 Variables in Path Expressions

Embedding variables in a path expression is a useful tactic. In fact a variable, enclosed in square brackets, can replace the selector (the first object reference of a path) or can be appended to any method invocation along the path's composition. The following examples help to pragmatically explore the usefulness of variables in path expressions.

The first example has the variable is bound to age value of employees having the same age as their manager in selected team.

```
team_ten.members.age[X] and team_ten.managed_by.age[x]
```

The second example will bind a manager object to the variable *X* only if the manager of the selected team has a salary greater than 20000. For each instantiation of the path expression an object is bound to the variable *X*.

```
team_ten.managed_by[X].salary >= 20000
```

The third example relates employees (through variable *X*) and their technical skills through variable *Y*. (Assume *ISA* is an infix relationship and **AND** a Boolean expression constructor). For each instantiation of the path expression objects are bound to the variables *X* (bound to employee) and *Y* (bound to skills).

```
[X].skills[Y] and [X] isa employee and [Y] isa tech
```

5.2.3 Heterogeneous set of objects

We have already stated that path expressions can return a set of heterogeneous objects through the deep extent relationship. The following expression does build a heterogeneous set and in general one expects the items not to be in any deep extent relationship).

```
team_ten.desc
```

This path expression, if evaluated in an appropriate data typing regime, returns all those objects that have a property called **DESC** from **TEAM_TEN**. In our example it would imply having the combined deep extents of classes **SKILL** and **TEAM**.

5.2.4 Physical Implementation of Path Expressions

A naïve implementation of path expression at data-access level would be to convert the expression into a cascade of identifier lookups and also create an index for each component of a path expression instance. This is expensive not only in terms of storing the excessively high number of possible paths in a class composition hierarchy but also in terms of maintaining these indexes up-to date. An access based on this naïve model would consume disk accesses in direct proportion to the number of components in a path expression. A number of valid proposals have been made to address indexing of path expression instantiations and the most significant are still references of Bertino's work on the Orion OODB prototype [BERTI89] and Valduriez seminal paper [VALDU87] on join indexes. In Bertino's work a number of indexes were proposed and the most popular two were the nested indexes and the path index.

In the case of a *nested index* a direct association is built between the selector object and the return object of a path expression. An index entry is made up of two values / objects references with the return object as the left most entry. This index is quite efficient for access paths that have an expression similar to the associations found in the path index.

Maintaining the index up-to-date if a large number of updates over an object collection materialise is a significant effort.

A *path index* is built by concatenating the selector object with the rest of the of path expression except the return value; the return value is used as the other partner of the index association. Updating this index is also an expensive operation but this structure is more flexible than path indexes as it can match against a larger number of path expressions. Therefore this index has a higher re-use potential.

5.2.5 Path Expression and Other Possibilities

Previously in the example of a path expression that is able to create a heterogeneous collection of objects that are not generally data-type compatible offered a glimpse of an interesting set of path expression – namely database schema exploratory expressions.

For example we might want to specify the starting attribute and the last attribute and allow the system to instantiate a minimum path between these two properties. In this case the variable is bound to a sequence of methods rather than actual instances. In fact we can therefore specialise the sort of the variables in a path expression to a sequence of either object instances, or object properties, or classes (this was introduced in Kifer's *et al.* [KIFER92A]). In this case there are two main techniques in place. First is the use of pattern matching (similar to regular expressions) to expand the path expression onto a schema's structures. Second is to use of deduction based on the semantics and composition of the schema (that include both implicit and explicit knowledge) to satisfy or infer a path expression

Another interesting short hand was proposed by Frohn *et al.* [FROHN94] that claimed an improved syntax and semantics of path expressions over those of Kifer *et al.* The authors propose the possibility of having "branches", called introducing the second dimension; in a path expression and consequently path expressions become more concise (view the two versions of the same path expression).

```
Remark:  Normal path expression
team_ten.members[X] and [X].age > 25 and [X].sex = 'female'

Remark:  Path expression with branches
team_ten.members.{age > 25 or sex = 'female'}
```

5.3 The Object Data Standard: ODMG 3.0

The *Object Data Management Group (ODMG)* was formed in 1991. Its participants were then a handful of people, led by Rick Cattell [CATTE94], interested in the development of object-oriented databases and their management systems. Their intentions were focused on the quick development of a set of specifications, loosely branded as a "standard", that would describe the data modelling and query modelling language primitives of interoperable object-oriented databases. Later ODMG aligned itself with standardisation efforts the *Object Management Group (OMG)*; specifically the data.

The standard's first publication came in 1992 and it was called "The Object Database Standard: ODMG – 93" [CATTE94]. ODMG justified the flaunting of "rigorous" procedures, as adopted with other standard's committees, on the premise that a specification could address two important aspects of reality at that point in time. The first was so that analysts and designers thinking of using an object-oriented database for a back-end would have peace of mind that the product chosen if ODMG compliant too different from the other contenders. The second was so that developers of OODBMS could concentrate on implementing the specification and improving on specific issues, like performance or data security. Quickly following the first publication versions 1.1 and 1.2 came. In the version 2.0 of 1997 Java binding API was added to the standard. Version 3.0 was published in 2000 [CATTE00] and the then board, now more numerous and having three types of members, assures us that version 3.0 should be with us for some time. The ODMG board claim on numerous occasions (e.g. on page 1) that theirs is a "de facto standard for this (OODBMS) industry". Indeed a good number of software producers claim membership to the ODMG. Also ODMG's web site had a page explaining what *ODMG compliant* and *ODMG certified* mean but no indication what the test suites are. It is not clear either when and up to what breadth are ODMG members and OODBMS producers implementing the ODMG 3.0 specification.

Although ODMG has ceased to exist, the Object Management Group in 2008 pledged to revive the standard with the 4th and next generation standard to include recent changes and realities of databases and computing. Consequently a group of academics and industry players, called the ODBMS.ORG, have also united efforts to carry on and distribute

research and products that aid ODG efforts. These have been attenuated, if not halted, after the financial crises of 2008.

5.3.1 The Generic Chapters

The *ODMG specification* is divided into a number of chapters (each with a specific scope). The main ones are the Object Model, the Object Specification Language, the Object Query Language, and a number of Language Binding chapters.

The *object model* on which ODMG's data and query models have their basis is a super-set of the *Object Management Group's (OMG)* object model [OMGRP08]. The additions to the OMG's *Interface Definition Language (IDL)* are artefacts that are typically present in databases but not in programming environments. ODMG believes that OMG's portability claim and industry wide support makes their object model a gratuitous lowest common denominator. Also OMG has eventually shown credence in part of ODMG specification; i.e. the Object Query Language.

The object specification language chapter deals with languages that are used to specify first the database structures, relationships, and constraints and second the up-loading and downloading of objects from the object database to the computational environment which is of no interest here. The first language is the *object definition language (ODL)* and must adhere to OMG's object model [OMGRP08].

The *object query language (OQL)* chapter presents a declarative language for querying. Along its development the structure and meaning of the retrieval query constructs have been respectively aligned with the syntax and meaning of SQL's SELECT statement [ANSIT86].

There are three chapters for language binding; one each for C++, Smalltalk and Java. Each chapter describes how to reconcile the host language data model with that of ODMG, write portable code (i.e. a repository is manipulated by any language that adheres to the binding rules), move objects to and from the repository and the programming language environment, and ways to invoke object queries. It also specifies what each binding should provide in terms of persistence (e.g. garbage collections and object locking) and functionality (e.g. transaction models).

5.3.2 The OMDG Object and Data Model

The core object model is used to represent the meaning of objects, object identifiers, object structures, intra-object relationships and methods. The principal constructs of the model are therefore:

A *literal* is a structured value complaint with the definition found in the previous sections;

An *object* has an identity in contrast to a literal and a structured value;

A *type* has a representation of a structure, of a set of constraints, and of a number of methods. A number of object instances can have the same type. The types specified for the design of an information system are the *user-defined types*;

An object's *state value* changes within the structure and constraints of the object's type. The state comprises attribute values and relationship instances;

A *method* implements part of an object's behaviour and it is qualified by the number and the type of arguments and the type of its result;

A *meta-structure*, or *database schema,* is a collection of object types and other features related to an OODB realisation and OODBMS mechanisms.

5.3.2.1 Types, Inheritance and Sub-typing

The OMDG's type has two aspects. The first is *specification* of its external characteristics (or properties) that are implementation independent. The specification includes attributes, relationships, methods and exceptions (i.e. error handler routines). This specification is realised by two language constructs: the **`INTERFACE`** and the **`CLASS`**. An *interface* definition predominantly specifies "abstract behaviour" only – i.e. for ODMG it is a method's signature. For each method we specify its name, its arguments and their respective type, and the return type of the method. A *class* definition specifies abstract behaviour (as just seen) and "abstract state" – i.e. methods, attributes, relationships, constraints and exceptions. In this context an abstract literal specification describes the state of each basic domain (e.g. Boolean, char, long, float, string) and the construction of values (e.g. collection, enumeration, union). There are structured abstract literals too (for example a 'date' data type).

The following is an example interface for **HOTELROOM**.

```
interface Hotelroom
    ( extent Hotelrooms
      keys   htl_roomnr, htl_roomnom
    ): persistent
    { attribute Unsigned Short htl_roomNr;
      relationship Set<Booking> is_booked_in_booking
           inverse Booking::is_for_hotelroom;
      attribute String htl_roomnom;
      attribute Unsigned Short number_of_beds;
      attribute Unsigned Short rate_per_night;
      };
```

The second aspect of ODMG's type is *implementation*. Each specification can have many and different implementations. For example the implementation includes a data structure which implements the abstract behaviour and a code segment to implement a method's semantics. It is important to understand the context of the language bindings here. These two aspects of types enable the ODMG to justify their claim that encapsulation is attained through their object model and programming language bindings.

In ODMG's object model inheritance is implemented through *sub-typing*. This relationship is regulated by two modalities. The first being that an object is also an instantiation of any ancestral type associated with its type. The second being that characteristics of the sub-type (e.g. methods and attributes) are modifiable, and the introduction of new property is allowed. The working standard does not provide for the purging of characteristics in the sub-type; so we have assumed that it is prohibited.

The inheritance (or sub-typing) relationship has two guises. The first manifestation being inheritance between interfaces (and classes) and the second is between classes. In the first case it is intended that inheritance affects only the abstract behaviour (i.e. methods) between interfaces only. Multiple inheritance is allowed but any eventual method name collision is to be resolved by the analyst. It is important to emphasise two points: first, these interfaces are not a complete specification (they could have missing attribute and relationship specifications); and second, interfaces cannot instantiate objects. The second manifestation of inheritance is the mechanism built-in with classes, called *extends*; classes inherit from an interface or another class. Only single inheritance is allowed in the 'extends' mode of inheritance. Classes do instantiate objects and implement their interface too.

If the designer needs to collect all instances of a type into an extent then she must explicitly associate an *extent* with a type. Because classes are the constructs that really instantiate objects then we expect to see an extent declaration tied to a class specification. Once an extent has been specified, and therefore the OODBMS is indexing the objects by their type, then it is also possible to specify "*value-based keys*" through the type's attributes. The key value, or values in the case of multi attribute key, has to be unique in the extent they are specified in. Also there could be a number of keys specified in a single extent.

```
interface Hotelroom
    ( extent Hotelrooms
      keys   htl_roomnr, htl_roomnom
    ): persistent
    { attribute Unsigned Short htl_roomNr;
      …
      };
```

5.3.2.2 Object Database, Object Creation, Objects, and "Collection Objects"

According to the working standard, an object-oriented database is the principal scope for all objects of an information system. In this database all objects are instances of a user-defined type that has been introduced to meet the requirements of the system. Closely associated with a database is ODMG's introduction of a *module* as a unit of declaration and specification of the database in which schema and application objects have a scope. A number of exception flags are specified for this module (for example **INTEGRITYERROR** and **DATABASECLOSED**).

An interface that is inherited by all other user-defined interfaces is the **OBJECT** interface. A few of the behaviours it bestows to all inheritors are the **COPY** (make a copy of the receiver object), the **DELETE** (purge the object from the collection unless referential integrity problems occur – in which case raise the **INTEGRITYERROR** handler), and the **SAME_AS** (to be explained shortly) methods. To create an object one must use the **OBJECTFACTORY** interface that provides the new method for this purpose. Note the language bindings would need to have their own implementation of the **OBJECTFACTORY** interface.

The ODMG specification of an object includes three parts. The first deals with identity, the second indicates an object's lifetime, and the third its value.

All objects have an *identifier*. This identifier is unique (with respect to the scope of the collection) and immutable. The standard bestows the responsibility of providing the

identifiers on the OODBMS. All interfaces and classes, as we have stated previously, respond to the **SAME_AS** method that returns a Boolean result on checking the identifier of the receiver object's to that of another object (passed as an argument) in the collection. Relationship instances use these identifiers.

Also the standard provides for *naming* of any object (over and above identifiers). These names are useful database entry points; but their management should not be taken lightly (both from the aspect of OODBMS design and from a DBA aspect). Homonyms are not allowed.

While it is obvious that a database stores objects for a period of time that is independent of the processes that create or access them (i.e. we are very much interested in object persistence) there are scenarios where some objects should purge when their process ends (the standard calls these *transient* objects). These transient objects are common in an application front-end and closely associated with the programming-language environment. The type of lifetime an object has, the standard wants us to understand, is independent of the object's type.

The standard makes use of the term *collection* to denote a number of distinct "elements" and therefore a type generator. There are some structural restrictions on the standard's collections. The most important restriction being that these "elements" must be of the same type – i.e. homogenous sets. If the "elements" are literals then the literals must be all of the same type. If the "elements" are user defined type instances then the objects must be of the same type. If the "elements" are collections of "elements" then the collections of "elements" must all be of the same type.

The specified behaviour of a user defined collection, which is a specialisation of the Object interface, includes **CARDINALITY** (returns the number of elements in the receiver collection), **CONTAINS_ELEMENT** (returns a Boolean flag if the identifier passed to the receiver is present in the collection), **INSERT_ELEMENT**, and **SELECT_ELEMENT** (returns an item from the collection that satisfies the query passed to the receiver collection). Some methods yield dynamic information on the collection's state (e.g. **CARDINALITY** and **IS_EMPTY** methods).

A useful companion to the collection interface is the *iterator* interface that allows access and traversal of a collection's elements. Other than the obvious methods **AT_END**, **RESET**, and **GET_ELEMENT**, there is an interesting Boolean method called **IS_STABLE** which indicates whether this iterator traversal is to be insulated from traversal processing side-effects over the same collection.

Collections can be specialised to a number of other particular collection types (for example *set*, *bag*, *list*, *array* and *dictionary*). For example the *set* is an un-ordered and collection of unique elements. The **SET** class (note the shift from interface to class here), which is a sub-type of **COLLECTION**, has a **VALUE** attribute whose type is parameterised with a type and a number of common set operations (e.g. **CREATE_UNION**, **CREATE_INTERSECTION**, **IS_SUBSET_OF**, and **IS_SUPER_SET_OF**). Some methods, for example the **INSERT_ELEMENT**, are incrementally modified to reflect the uniqueness semantics of set over those of a collection.

5.3.2.3 Literals

The standard specifies a good number of types whose values do not have an identifier; these we have seen are called literals. These types are an exact equivalence of OMG's IDL own values. The most basic – called *atomic*, and very close to our basic domains of the previous section, include the following list; *long*, *long long*, *short*, *unsigned long*, *unsigned long*, *unsigned short*, *float*, *double*, *Boolean*, *octet*, *character*, *string* and *enumeration*. The last is really a type generator as a definition is required to specify the list of literals making up an enumerate type. For example to specify sex type one could define the enumerated type gender with the literals "male" and "female".

Another set of literal type generators is the collection of literals (with the same variety as the collections just presented but restricted to literals).

We can also use the record data type generator (the standard calls these structures). Each *structure* has a name and a fixed number of components, with each of these having a name and a literal data type. A text book example being an address with its components street, city and postcode. There are a number of structures pre-defined too: e.g. date and time.

Since literals do not have an identifier then the use of **SAME_AS** is not compatible. To compare two literals one uses the **EQUALS** method. The standard lists a number of rules, related to the data type that gives the exact meaning of two literals being equal. This is based on structure and values. An almost verbatim reproduction of a selection of these rules follows.

Two literals, ***x*** and ***y***, are considered equivalent if they have the same literal type and:

- ✓ are both atomic and contain the same value;
- ✓ are both sets, have the same parameter type ***t***, and each element of ***x*** is in ***y*** and each element of ***y*** is in ***x***;
- ✓ are both structures of the same type and for each component ***j*** then ***x.j*** and ***y.j*** are equivalent.

5.3.2.4 Data Typing and Type System

The ODMG claim that their model is "strongly typed" and the standard explains that consequently all objects and literals have a type and every method has its arguments and result typed too. Two types are equal or compatible according to the following paragraph from the standard.

> "Two objects or literals have the same type if and only if they have been declared to be instances of the same named type. Objects or literals that have been declared to be instances of two different types are not of the same type, even if the types in question define the same set of properties and operations. Type compatibility follows the sub-typing relationship defined by the type hierarchy. If ***TS*** is a subtype of ***T***, then an object of type ***TS*** can be assigned to a variable of type ***T***, but the reverse is not possible.
>
> Two atomic literals have the same type if they belong to the same set of literals."

It is very important to note that significant work has been done to iron out issues with the type system and type inference required for expressions in this standard especially for queries and language binding API. Two papers that stand out are Alagic's work [ALAGI99] and Bierman and Trigoni [BIERM00].

5.3.2.5 State Properties

The *attributes* and *relationships* of an object are typically defined through a class construct. Each attribute is associated with a data type and each relationship has a cardinality qualifier and an indication (through a mention of a class) of which instances can satisfy it. While literal values are "copied" for each attribute's value slot, relationships and attributes whose values are instances of a class have the instance identifier bestowed on them.

In some examples found throughout the standard one finds use of a *readonly attribute*. This attribute qualifier is indeed a legal construct according to the standard's ODL grammar. An exact meaning of this qualifier is found in references on the IDL language (e.g. Baker's introduction to CORBA in [BAKER97]). A **READONLY ATTRIBUTE**'s value is only available for access (for example the language binding implementation will not provide a setting method for the attribute). None the less other methods are able to access and change the value of a read-only attribute.

If an "attribute" is considered to be part of an abstract behaviour then an interface can have both attribute and relationship definitions. One definitely has to respect the standard's requirement that interfaces must be non-instantiating constructs. The general syntax is:

```
ATTRIBUTE < TYPE | CLASS > < ATTRIBUTE NAME >;
```

Relationships are specified with the **RELATIONSHIP** construct and require some more explaining in terms of the ODMG specification. The general syntax is:

```
RELATIONSHIP < TYPE CONSTRUCTOR > < RELATIONSHIP NAME>
 INVERSE <RELATIONSHIP NAME>;
```

Firstly relationships are only binary. These are more "expressive" than the ERM's binary ones because the type constructor (which is a singleton or normal set in the ERM) could be any of ODMG's collection data types (e.g. a list type constructor implies order). Secondly each relationship is annotated with backward traversal information (i.e. the standard calls these *traversal paths* and uses the **INVERSE** construct to implement these) and therefore each binary relationship is specified in two classes (or interfaces) that participate in it. The following code constructs show how an 'employs' 1:N relationship between team and employee is specified.

```
Reamrk:  code template to implement the one-to-many relationship between
           team and employee

CLASS team {
    Remark:  other specifications
    RELATIONSHIP SET<employee> employs
    INVERSE employee::employee_of;
};

CLASS employee {
    Remark:  other specifications
    RELATIONSHIP team employee_of
    INVERSE team::employs;
};
```

In ODMG's relationship construct the many part is resolved by using the set data type constructor with the data type parameter being the class of the objects the relationship is qualified with. The one part is resolved by using the class of the objects the relationship is qualified with. In the case of the M-N relationship each side of the relationship would have a set data type constructor qualified with the name of the opposite class (in the relationship).

It has to be noted that ODMG relationships are binary and take no attributes.

The standard requires the OODBMS to maintain referential integrity for all relationships declared through the **RELATIONSHIP** construct. If an operation jeopardises the referential integrity then the **INTEGRITYERROR** exception is raised. If there is an attribute definition whose domain is in another class instance then the OODBMS is not responsible for maintaining its referential integrity; consequently the application developer would need to write in the referential-integrity code.

The OMG's IDL does not have the concept of a relationship. We have already remarked that ODMG object model is conformant to OMG's IDL, therefore the **RELATIONSHIP** construct is mapped onto a sequence of IDL constructs (i.e. read-only attributes, methods to form and maintain a relationship, and exception flags).

5.3.2.6 Operations & Exceptions

Each type has a range of behavioural aspects associated with it. Each of these is implemented by a method. And each method has a name, type signature for its arguments and return values, plus an explicit indication of which exceptions the method's implementation may raise during its execution. Also each argument may have a qualifier (e.g. **IN**, **OUT** and **INOUT**) that indicates whether the value of the argument is read only,

return only, or read and return. The programming-language bindings provide the language through which the implementation details are coded to provide these aspects of the method's signature. The following examples show two methods. The first method **EXPECTED_BONUS** returns a sum (of type float) of an employee given his performance as expressed in the 'level' argument and other instance details. If the method has insufficient details a named exception is raised and consequently an exception handler can cater for the occurrence. The second method, **RETIRE**, requires a termination date for the computation of the behaviour in the 'end_date' argument. An employment-termination process is a contrived procedure and therefore a number of distinct exceptions might be flagged. Methods that don't return values are qualified with the **VOID** keyword.

```
Remark:  part of an interface for employee
Remark:     describing methods
CLASS employee
   (EXTENT employees)
   { Remark: some other properties
     float expected_bonus(IN short level)
     RAISES (bonus_not_applicable);
     VOID retire(IN data end_date)
     RAISES (employee_already_gone, employee_retire_date_incorrect);
};
```

There is no syntactic distinction between methods that have an effect on a database state and those that don't.

The following example shows how the **EMPLOYEE_ALREADY_GONE** is defined and adorned with properties (i.e. the employee's already recorded date of termination). The properties of an exception are a concise approach for passing values between the exception raising the computation and the error handler.

```
Remark:  part of an interface for employee
Remark:    defining exceptions
CLASS employee
   (EXTENT employees)
   {  Remark some other properties
      EXCEPTION employee_already_gone
      {date when};
};
```

There is another aspect of methods that needs telling. The ODMG's claim that "the object model assumes sequential execution of operation" (i.e. in section 2.7 of [CATTE00]). Consequently the object invoking a method is blocked until the receiving object has completed its execution. Yet a look at the relevant part of the ODMG's ODL grammar one finds the **ONEWAY** method definition qualifier. If an object calls a message to a "one-way"

method then the calling object is not blocked (according to OMG's IDL a one-way method should be termed as "a best effort semantics of message passing"). Also such one-way methods have structural restrictions, specifically no return types, no OUT and INOUT arguments mode and no exception raising capabilities. This explanation is completely missing from ODMG's object model specification.

5.3.2.7 Data Dictionary (or Metadata Hierarchy)

The application program's data requirements in a database environment are recorded in the schema definition, as we have seen in the first chapter. The ODMG calls this the *metadata* and it is stored in the ODL Schema Repository. The ODMG metadata is a superset of ODM's IDL Interface Repository because of the richer "structures" (e.g. relationships) found in it. The standard specifies the interface of each constituent part of the metadata and provides a terse textual description of the properties and methods. The exact semantics of the operations are missing.

There are thirty-two interfaces with thirty inheritance relationships between them. Most of these inheritance relationships are of the single-inheritance type but four rely on multiple inheritance. The longest depth of the *ISA* relationship is of four (e.g. **`DICTIONARY`** / **`COLLECTION`** / **`TYPE`** / **`METAOBJECT`** / **`REPOSITORYOBJECT`**).

Rather than giving a sequential description of each interface we are presenting the data dictionary in a different perspective; namely as a description of a database. This description is found in Appendix – ODMG Data Dictionary attached.

5.3.2.8 Concurrency Control and Transactions

Database updates change its state. In operational databases this is a principal activity. An important abstraction used to describe this activity is the transaction model (a number of related retrievals and updates are grouped as a logical whole) and there are a number of basic principles that guide the meaning and implementation of this model (a common group of principles are the *ACID* – Atomicity, Consistency, Isolation and Durability). Each transaction model's implementation's correctness, safety, and efficiency are of fundamental importance to all operating databases (our bible here would be Gray and Reuter [GRAYJ92]). Each DBMS (and even transaction processing monitors) do come with their own

implementation variants to cater for different manifestations of sharing of data and automated recovery from failure.

OMG categorically state that all updates must be through a transaction. The standard does not specify the type of transaction model (e.g. nested transaction model) but emphasises that serialisability has to be upheld within the ACID context. If the implementers are interested in distributed transactions then the standard expects the implementator to adhere to the ISO XA code standard in respect to this issue.

Transactions are objects for the ODMG standard. To create a transaction a **`NEW`** message needs to be sent to the implementation of the **`TRANSACTIONFACTORY`** interface (we need to recall that factory interfaces are host programming language implementations). The transaction becomes an instance of the **`TRANSACTION`** implementation and can respond to a usual variety of transaction processing operations. These include **`BEGIN`**, **`COMMIT`** and **`ABORT`**. As already stated each update has to be done through a transaction and therefore these changes are to be made through it, consequently these updates operations must first obtain an association with a transaction instance.

Our description of ODMG's transaction processing methods has been a high-level one and would leave a wide range of implementation possibilities. This is not exactly so because the standard states that the object model uses a locking based approach. The type of locks it requires are the typical "read", "write" and "upgrade" locks. The standard also gives brief descriptions of implicit and explicit locks. The duration of locks is associated with the transaction termination state (i.e. committed or aborted). The ODMG committee stress that the lock's semantics and lock's duration are taken from other standards. Namely OMG's Concurrency Control Service and SQL-92 definition of transaction isolation level 3 [CANNA93] respectively. Unfortunately the link between transaction serialisability and isolation levels is not obvious (see Berenson *et al.* [BEREN95]).

5.3.2.9 Query model & Path Expressions

OQL uses the structures and relationships created by the ODL and applies the same type checking for object expressions too.

Path expressions are made from a sequence of attributes and relationships emanating from either a named object or an object identifier. Two types of results can be returned by a path: either a scalar object, or a collection of objects. The data type of a path is determined by its last attribute. There is a restriction on the construction of a path expression namely that a collection type (e.g. set, bag, list) can only appear at the end of a path.

OQL's query specification follows SQL's "SELECT FROM WHERE" structure. The FROM clause is made up of a sequence of object collections and has a facility for renaming their instances. Object collections are either a collection path expression or a class. The WHERE clause is a logical predicate that determines which objects from the named collections are passed to the output–that is the SELECT clause. The WHERE clause allows for nested queries, existential, and universal quantification queries. The WHERE's predicate is built from a number of Boolean returning conditions and composed through logical connectives (e.g. AND, OR). The SELECT clause, in its simplest form, returns a bag of structures and the structure must have at least one attribute. If the query requires other type constructors, like set and list, then the SELECT clause requires additional qualifiers. For example to return a set of structures the DISTINCT qualifier is added to the SELECT clause. For a list of structures one adds an ORDER BY < attribute name + > qualifier.

Aggregate queries are possible and follow the GROUP BY HAVING syntax found in SQL.

5.3.2.10 Databases

The ODMG standard uses an instance to describe the whole database (i.e. a database instance behaviour is described in the **DATABASE** interface and an instance is created through the new operation of the **DATABASEFACTORY** interface). The database interface is specified by the standard and includes the database administrator commands *start-up* and *shut down* operations (these are called, somewhat confusingly, **OPEN** and **CLOSE**). No database access is allowed unless the database is open and shutdowns are allowed if no database transaction is in progress. Although admirable, pragmatically this is not usually a good idea. An important set of operations caters for object naming of the database instance (these are the **BIND**, **UNBIND** and **LOOKUP**). An important method in the database interface is the **SCHEMA**. The **SCHEMA** operation associates the database instance with its schema "root" object (i.e. an instance of the module in the data dictionary).

5.3.3 Critique of ODMG's Object Model and ODL

One basic problem with the standard was that it was a long time in coming. Worst still was the timing of the better version 3 publication (year 2000) as it coincided with a cooling down period for object-oriented databases.

Another gross disadvantage, and an unfortunate situation, is the standard's unclear description in a number of parts of the text. This is unfortunate as object-orientation themes did not have an unambiguous nature both in academia and in industry. The standard has a large number of missing features (as we have seen in data modelling). This is obvious in other specialised areas as: views, data security, integrity constraints, and triggers are totally consigned to the 'implementation features and options' of the vendors.

The standard needs a good polishing up in terms of the transaction model too. There are some problems in the data typing of expression and seriously affect the structure and meaning of query expression, for example.

Ironically the companies that were on the standard committee have not adopted much of the standard other than the programming-language mappings.

5.3.4 EyeDB

Object databases and their management tools had started with a number of research prototypes. These include Orion, Iris, Postgres and O_2. Currently the stronger and commercially available object databases are from Actian's Versant {WWW.ACTIAN.COM}, Objectivity {WWW.OBJECTIVITY.COM} and Progress Objectstore {WWW.OBJECTSTORE.COM}. All of these claim their success is based on adoption in niche application domains and in their performance. Performance is based on writing queries (e.g. C++) over their object stores

In these last ten years a number of store management tool sets for objects have emerged but although they offer persistence they all lack any adherence to ODMG data model and language bindings. Examples include: MongoDB {WWW.MONGODB.ORG}, db4o {WWW.DB40.COM}, Caché {WWW.INTERSYSTEMS.COM/CACHE}, and CouchDB {WWW.COUCHDB.APACHE.ORG}.

Another interesting option albeit lacking presence, and one adopted in this research is called EyeDB; with Genethon Laboratories {WWW.GENETHON.FR} being the initial developers. The object database was required for the Genome project. EyeDB is still very strong in the

bioinformatics community. The current version is effectively its third rewrite and is ODMG version 3 compliant [VIERA99] with ODL and OQL. It provides an extensive object model with inheritance, collections, methods, triggers, and constraints. It also offers language bindings to C++ and Java and has a scripting language that codes the object database methods and triggers.

EyeDB has been made available as open source since 2006. Support is available mostly through direct communication with authors.

5.3.5 ODL & OQL Interactive Session

The following is a sequence of ODL, OQL, scripts for an EyeDB-based object collection. The artefacts were executed on EyeDB CLIs.

```
Remark Define a class called person
class Person {
        attribute string fname;
        attribute set<tel*> telno;
        …
        relationship set<Project*> workson
             inverse Project::staff;};

Remark Define a class called student which is a subclass of person
class Student extends Person {
        attribute string stage;
        attribute set<result*> results;
        relationship Course* enrolon
             inverse Course::candidates;};

Remark Define a class called lecturer which is a subclass of person
class Lecturer extends Person {
        attribute integer salary;
        … };

Remark Define a class called course
class Course {
        attribute string cname;
        attribute integer cnum;
        … };

Remark Define an enumerated list for gender domain
enum gender {
        male   = 1,
        female = 2 };

Remark Run a script to generate test objects for course
Remark (? Is command prompt)
? for (x in 1000 <= 1999)
{ varname  := "course" + string(x);
  nw  :=
  " := Course(cname:\"name"+string(x)+ "\", cnum:"+ string(x)+ ");";
   cmd := varname + nw;
   eval (cmd); };
? \commit

Remark Extent and deep extent queries
Remark (= is response prompt and – start a remark)
-- Includes instance of person class and
-- all its subclasses

? count(select x from Person x);
= 1390

? select x.class.name from Person x
   where x.fname = "fn 6002";
= bag("Student")
```

```
-- Find Person instances that are instance
-- of class Lecturer
? select x.fname from Person x
  where classof(x) = "Lecturer";
= bag("fn 5025", "fn 5022", …

? count(select x from Person x
   where classof(x) = "Person");
= 350

? count(select x from Person x where classof(x)="Lecturer");
= 10
```

Remark Simple queries to check test objects

```
-- count is an aggregate function
? count (select Course);
= 1000

-- course10 is a variable points at an object
? select course10.cname;
= "name10"

-- first is function that returns 1st
-- instance of a collection
? first(select Course.cname);
= "name183"
```

Remark Controlling the output data type

```
-- return a bag of …
? select Student;
= bag(8831.3.17414:oid, 8803.3.1005572:oid,…

-- return a set of …
? select distinct Student;
= set(8831.3.17414:oid, 8803.3.1005572:oid,…

-- return a list of …
? select s.fname from Student s
  order by s.fname;
= list(8831.3.17414:oid, 8803.3.1005572:oid,…

-- return a set of structures
? select distinct
  struct(student:s.fname,level:s.stage)
  from Student s;
= set(struct(student:”stud 11”,level:”s”),…
```

Remark Selection Queries with path expressions

```
-- conjunction
? select p.fname from Lecturer p
  where p.addr.town="town 511"
    and p.salary > 45000;
= bag("fn 5050", "fn 5037")

-- disjunction
? select p.fname from Lecturer p
  where  p.addr.town="town 511"
     or  p.salary > 45000;
= bag("fn 5050", "fn 5011", … "fn 5037", "fn 5041")
```

Remark Projection query

```
? select bag(s.fname, s.enrolon.cname)
  from Student s
  order by s.fname;
= list(bag("fn 5048",”f”), bag("fn 5112",”f”), bag("fn 5260",”f”), …
```

Remark Implicit join query

```
-- find which students and persons share
-- the same town but not name (the last
-- condition rids us of spurious instances)

? select struct(cons:s.fname,
         emp:p.fname, town:p.addr.town)
 from Student s, Person p
 where s.addr.town=p.addr.town
   and s.fname != p.fname;
= bag(struct(cons: "fn 4013", emp: "fn 4000", town: "town 500"),
struct(cons: "fn 4006", emp: "fn 4019", town: "town 506"), …
```

```
Remark Query over data dictionary – direct subclasses of class Person
? select struct(cn:x.name,cp:x.parent.name)
  from class x where x.parent.name="Person"
= bag(struct(cn: "Student", cp: "Person"), …)

Remark Stored procedure: recursively traverse the class hierarchy for
subclasses
-- definition
  function subclasses(cn) {
  clist  := bag();  cclist := bag();
  cmd := "clist:= (select x.name"+
  " from class x"+
  "where x.parent.name = \"" + cn + "\");";
  eval cmd;
  if ( clist[!]=0 ) { return clist; }
  else
  { cclist := clist;
    for ( c in clist )
    { cclist += subclasses(c); }
  return cclist;  }};

-- invocation of function
? select subclasses("Person");
=bag("Student", … )

Remark Stored procedure: generate a random digit
-- definition
function rnddigit() {
  dseq := 0;
  dseq :=
   (select time_stamp::local_time_stamp()).
   usecs;
  s := string(dseq);
  l := s[!];
return s[l-1]; };

-- invocation of function
? select rnddigit();
= '9'
```

5.4 Summary

Interest in object-oriented databases started in the early Nineties with much diversity in the data models and clearly an alignment effort was required. In this chapter we have synthesised a data model that provides a basis for an object-oriented database with a practical and effective selection of features. Our object-oriented data model has classes, *ISA* relationship, data types, and objects composed of value, identity and instance-of relationship. We emphasise that there are two separate relationships from classes to inheritance and data types.

A section was dedicated to path expressions; these are constructs that abbreviate object navigation expressions. Path expressions implement what are called implicit joins over the database schema. Advanced path expression can be adorned with universal quantification and pattern matching.

The standardisation process by ODMG finished in the early Naughties; its object and query model was a focal point in this chapter. Their final version is a good standard. We have

given a critical look at its data model and how it is used as a basis to its query model. The main critical points were: it took a long time to produce; its quality took too long to pick up; although adequately thorough in its object and query model it lacks precision and completeness in substantial parts (e.g. views, integrity constraints). Also some features, for example relationships, need more and wider capabilities.

EyeDB is an open-source OODBMS and has a good level of compliance with ODMG's ODL and OQL. This level of conformance is hard to find and was the main reason for its adoption here. EyeDB is used to read and creates schemas generated from a framework we developed that reads an EERM and create a sequence of ODL and EyeDB constructs (e.g. integrity constraints not catered for in ODMG) to implement the design.

During the progression of this chapter a number of side references were made to other technologies that can be considered to have a basis in one of the object-oriented database features presented; e.g. Xpath and path expressions.

Nonetheless we still have to resolve a basic requirement made earlier on; i.e. to have a declarative and a procedural language that operates on our synthesised object-oriented data model. In the following two chapters we show how a deductive database caters for the declarative query modelling; furthermore the same logic can describe an array of structural constraints. Once the logic is in place we can present our framework and an algebra that operates on an object-oriented database.

Chapter 6

Deductive & Object-Oriented Databases (DOODs)

6 – Deductive & Object-Oriented Databases

We have defined a class-based object-oriented data model we need a calculus that can be its basis. Since the eighties a large body of work developed about adding logic programming to the relational model; i.e. Datalog. These databases have been named as deductive databases. Through Datalog a number of important results have been established: these include methods of evaluation, query complexity analysis, and algebraic equivalence. Furthermore various "additions" to Datalog, for example 'function' symbols and 'logical' identifiers, were proposed.

A logic that has attracted our attention is called F-logic [KIFER95]; it offers first-order semantics, a number of axioms for object-oriented features, and the possibility of logic programming. The implementation of F-logic adopted here is Flora-2 [YANGG08].

Another important requirement for the calculus selected is to represented and enforce database-integrity constraints which are described in the chapter seven.

6.1 Deductive Databases and Datalog

It was indicated that the relational data model comes with a number of query languages. In particular one example is the relational algebra, which is a "constructive" equivalent of relational domain calculus and tuple calculus. Either calculus is a declarative language. These two declarative languages are a sub-set of first order-logic that has been in use for many years.

As regards our synthesised object-oriented data model we are yet to introduce a "logic" for it. This is an important aspect as it will enable us to describe and specify 1) database structural and behavioural rules, 2) access patterns to the database (e.g. queries), 3) programs. There is really a shortage of a logic that caters for an object-oriented data model in the same way as relational calculus caters for the relational model. Despite some work in this area, there is still a need to develop a query logic that truly captures the flavour of the object-oriented paradigm. Broadly speaking this area came to be known as *deductive object-oriented databases* (DOODs) – clearly trying to create a synergy between deductive databases and the object-oriented paradigm.

During the nineties there have been a reasonable number of proposals and even prototypes of these logics but no clear "winner" has materialised. If we had to describe the proposals at a meta-level the spectrum spreads from extending a known model (e.g. relational) with object-oriented features, to axiomisation of object-oriented themes into the semantics of the language. Each strategy has its advantages and negative aspects.

The proposal adopted here comes from Kifer, Lausen and Wu and is called *F-logic* (or Frame Logic) [KIFER95]. F-logic can be placed at one extreme of the above spectrum; i.e. F-logic has object-oriented data modelling features entrenched in its formal semantics. Major influences on F-logic are from Ait-Kaci [KACIA94], Maeir [MAEIR86] and HiLog [CHENW89]. Parts of F-logic have been implemented in a number of systems, and in particular in two projects. The first is called Florid [MAYWO00] from the University of Freiburg, and the second is called Flora-2 from the Stony Brook University. Other than the database field, F-Logic has overlaps with Artificial Intelligence.

6.1.1 Logic and Databases

In the very first sentence of Gallarie and Minker's seminal compilation on logic and databases [GALLA78] they state "Mathematical logic has been applied to many different areas, including that of databases". Indeed the authors project this classical reference, based on the then recent development of relational logic, on database related problems. In a later compilation by Minker [MINKE87], the editor explains that automated theorem proving not only unifies but also propels databases and logic programming. One of the differences between these two references is the enabling of database design with computational programs (i.e. expressed in logic). Effectively *deductive databases* became a reality and have been present with us since then.

6.1.2 Logic Programming

The high interest and activity in automated deduction system culminated in the 1960s with Robinson's [ROBIN65] publication of the *resolution rule* as a sole rule of inference. Consequently the inference rule is well suited for a computerised system.

An excellent example of the workings of the resolution rule is found in tutorial hosted at the following URL { WWW-RCI.RUTGERS.EDU/~CFS/305_HTML/DEDUCTION/RESOLUTIONTP.HTML } maintained by

Charles F. Schmidt. For a given first-order knowledge base it proves, through iterative resolution, that "Marcus hates Caesar".

Shortly afterwards Kowalski [KOWAL74] and Colmerauer [COLME96] introduced logic programming whose inference system is based on the resolution rule. The first logic-programming prototype was conceived at Marseille by Colmeruaer and Roussel; this prototype is the basis of Prolog.

Kowalski described a logic program as a collection of *clauses*. These clauses are usually a sub-set of first order predicate calculus sentences (or rules) and have the following form:

$$A \leftarrow B_1, B_2, \ldots, B_n.$$

The symbol A is called the *head* of the rule, and the B_i's are a conjunction of predicates– collectively called the body of the rule. The meaning of the rule is straightforward: if an instantiation of the body's predicates through the rule's variables evaluates to true then as a consequence we have established the truth of the head predicate.

Each of these clauses can have a procedural interpretation. A program is invoked through a *goal* (a rule without a head):

$$\leftarrow G_1, G_2, \ldots, G_k.$$

Each clause in the goal (i.e. G_i) is similar to a procedure call. If we are evaluating G_j then we need to find either a predicate (a rule without a body) or a rule with the same name, unify (a process that passes argument values or selects data), and continue evaluating the goal. Note that, if the goal predicate G_j is to be unified with A above, and the 'new' goal is:

$$\leftarrow G_1, G_2, \ldots, G_{j-i}, (B_1, \ldots, B_n)\theta, G_{j+1}, \ldots, G_k.$$

Remark: θ is the unification substitution.

The evaluation terminates when the empty goal is produced. Typically the logic programmer specifies what the problem is and the 'in-built' control specification solves the program. Although most logic programming control is through resolution theorem proving, there are some other alternatives. The realisation of Kowalski's "Logic + Control = Logic Programming" was for all to see [KOWAL79].

6.1.3 Deductive Databases

Once logic programming became a reality and the relational data model had an established formal foundation then it was inevitable that deductive databases would become their cornerstone.

Furthermore it was immediately apparent that the relational query language could not describe basic database queries, for example recursive union, whereas the logic programming stream does oblige. Finally some researchers claimed that the use of one language (i.e. for data and query modelling and procedural computation) would address the nagging 'impedance mismatch' problems–Maier's work is credited with the introduction of the term [MAEIR86].

For the record we need to introduce the meaning of a deductive database. A *deductive database* is a collection of clauses. These clauses are first order predicate calculus formulae. A clause could describe a fact, a rule to derive or compute data, or an integrity constraint. Facts are typically called the *database extension* (EDB), while derivations through rules are called the *database intension* (IDB). There is actually nothing different in the structure of logic programs and deductive database clauses. At this point one has to substantiate the differences between deductive databases and logic programming.

6.1.3.1 Differences between DD and LP

These differences are not absolute, but do indicate the relative placing of the two techniques.

- ✓ In deductive databases the number of facts heavily outnumbers the rules. This isn't in general the case in logic programming. Consequently there is a shift in the optimisation techniques implemented in each respective field (e.g. in databases the management of data is predominant while in logic programming the efficiency of inference takes the first priority). Yet another facet of this is that in databases one tends to map different physical implementation (e.g. indexes) for items having different characteristics (e.g. access patterns or integrity constraints).
- ✓ In deductive databases the role of integrity constraints is significant. In logic programming these techniques are almost non-existent.
- ✓ The structure of the response to goals is different. In deductive systems one expects the set at a time method while in logic programming one expects an instantiation at a time method (i.e. the result of a resolution-based refutation procedure).

- ✓ In databases, the goals (or queries) are typically meant to find patterns that actually match, rather than all possible candidates that can be generated to match.
- ✓ In logic programming we expect to use certain artifacts, e.g. functions and incomplete data structures, freely whereas these are not generally allowed in deductive database languages. The implication of this is profound: logic programming can compute more queries than deductive database languages (e.g. known as Datalog).
- ✓ When executing programs over a deductive database the underlying schema also determines what the programs mean. In logic programming the set of constants, functions, and predicates in the program suffice.

6.1.4 Datalog

One of the most researched languages in deductive databases must be Datalog; a term introduced by Maier and Warren in [MAIER88]. Basically Datalog started as a simplified version of Prolog with database friendly features. Datalog is a rule-based language for relational predicates (i.e. value based), but functions are not allowed in arguments which is not the case in Prolog. The semantics of Datalog programs is formalised and well defined. Since its inception Datalog has been used as a yardstick against which to compare in terms of theoretical properties and also as a reference language to state significant results.

We therefore need to describe the language, associated evaluation techniques and some results. This exposition is based on Ullman's textbooks ([ULLMA90]) and Ceri, Gottlob, Tanca's survey [CERIS89].

A good and useful implementation is DES (Datalog Educational Systems) by F Perez of Universidad Complutense de Madrid { WWW.FDI.UCM.ES/PROFESOR/FERNAN/DES/ }.

6.1.4.1 Datalog – the language

A simple program in Datalog is (line numbers on the left are not part of Datalog):

```
1.    anc(X, Y) <- parent(X, Y).
2.    anc(X, Y) <- anc(X,Z),parent(Z, Y).
3.    parent(X, Y) <- father(X, Y).
4.    parent(X, Y) <- mother(X, Y).
5.    father(eric, mario).
6.    mother(alice, simon).
7.    mother(alice, mario).
```

Lines 5, 6 and 7 denote facts with the last line relating 'alice is the mother of mario'. Lines 1, 2, 3, and 4 have rules through which other facts can be inferred. Rule 4 says that if 'X is the mother of Y' then 'X is a parent of Y'. Note that the scope of variables is the rule and the multiple occurrence of a variable in a rule implies that the same constant is substituted in each occurrence.

A query is an expression; the "**?-**" symbol is used prior to the query as a command prompt. For example the following tries to find the father and mother of objects.

```
?- mother(X,C), father(Y, C).
```

A possible response to the query would be X=alice, Y=eric and C=mario.

6.1.4.2 Syntax

The basic elements of Datalog syntax are constants, predicates, variables (starting in uppercase), and terms (i.e. a variable or a constant). Also the usual logical connectives (e.g. 'and', 'or', 'not', and 'implication') and quantifiers (universal and existential) are used.

A Datalog *term* is a constant or a variable.

A Datalog *well formed formula* (i.e. wff) is defined inductively as:

- If p is a *n*-ary predicate and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is an atomic wff.
- If A and B are wffs then so are the following:

```
not A
A and B
A or B
A <- B
A -> B
A <-> B
```

- If A is a wff and X is a variable then so are the following:

```
∀ X (A)      Remark: X is bound within A.
∃ X (A)      Remark: X is bound within A.
```

Those wff that contain no variables (e.g. facts) are called *ground* wff. A wff is *closed* if every variable in the formula is quantified. Conversely if there is a variable in a wff that is not quantified then the wff is *not closed*.

A *clause* in Datalog is a closed wff of the following form:

```
∀ X1, … , ∀ X2 ( A1 or … or Ak or not B1 or … or not Bl )
                Remark: where Ai and Bi are atomic wffs.
```

A *definite clause* is a Datalog clause with one positive atomic wff and zero or many negative atomic wffs (another name for these is Horn clauses). In this structure called a *rule* the

positive clause is the head and any negative wffs form the body. A rule with an empty body is sometimes called a *unit clause* and a unit clause without any variables is a *fact*. A *definite clause program* (*or positive Datalog logic program*) is a collection of definite clauses.

6.1.4.3 Semantics

It is interesting to note that there are three different, but equivalent, ways on how to describe the semantics of a definite clause program. These are the model theoretic, the proof theoretic, and the operational. (Ullman in [ULLMA90] mentions also the *ad hoc* semantics (e.g. like Prolog) but quickly loses interest in it). In essence the model theoretic provides a declarative meaning to a program and the operational semantics provide a coupling between a program's evaluation (i.e. bottom-up) to deductive databases. These are the two semantics descriptions of interest in this study. Out of completeness, the proof theoretic interpretation is related to the SLD-resolution and top-down program evaluation.

The *model theoretic semantics* often called Tarskian semantics has two main aims: the first is to enumerate the objects (or individuals) comprising the domain of discourse and their interrelationships; and the second is a mapping from a language's symbols to those objects in the discourse. Consequently the semantics of the language depends both on the world one is trying to represent and on how the constants and predicate symbols in the syntax correspond to individuals and properties of the world. This is the *interpretation*. We are interested in interpretations that make a Datalog program true; in which case the interpretation is called a *model* of the program. To give a formal cladding to these ideas the following steps are required.

Given that L is our language (e.g. Datalog) and P is a positive Datalog program then we initially select a non-empty set of elements U, called the *domain of interpretation*. Then an interpretation of L is defined as:

1. For each constant in L, an assignment of an element in U.
2. For every predicate p in L, an assignment of a mapping from U^n into {TRUE, FALSE}

   (where n is the arity of p).

It is important to state that for definite programs it suffices to assume that constants represent themselves in the interpretation (an observation due to Lowenheim, Skolem and Herbrand) – these particular interpretations are called *Herbrand Interpretations*. The

Herbrand universe for L, denoted U_L, is the set of all terms that can be built from *L* (if *L* is devoid of constants then one introduces a single constant). The *Herbrand base of L*, denoted by B_L, is the set of atoms that can be generated by assigning objects of U_L to the arguments of predicates found in *P*.

For a Datalog program *P*, the Herbrand universe, *Up*, and the Herbrand base, *Bp*, are respectively, defined as U_L and B_L of language *L* that has constants and predicates identical with those appearing in *P*.

It quickly becomes apparent to us the astronomical number of possible interpretations. In our example, with just four predicates (with each having two arguments), and four constants, the size of B_p is 64 (i.e. 4 * 4 * 4). Since any subset of B_p is an interpretation, then there are 2^{64} Herbrand interpretations.

Let us now consider the ground instances of a rule, e.g. *p*, in *P*. If *ground(p)* represents the ground instances of a rule *p* (obtained by assigning constants from U_B to the variables in *p*) then the set *ground(P)* denotes all the program's predicate ground instances assigned constants from U_B.

Given the enumeration of *P* through *ground(P)*, then one is able to check whether the elements of ground(P) are in an interpretation of *P*. If the instance is in the interpretation then the ground atom is said to be *satisfied* otherwise it is not. An interpretation that satisfies all rules in *P* is called a *model* for *P*. The following interpretations for the program used earlier gives a tangible idea of the definitions here.

```
Remark:interpretation one
       I1 = { father(eric, mario),
        mother(alice, simon),
        mother(alice, mario)}.

       Comment:- Is I1 an interpretation?  Rules 5, 6 & 7 are
       alright.  But what about 4?  The body is in this
       interpretation but the head is not.  Therefore I1 is not an
       interpretation of the above P.
```

```
       Remark: interpretation two
       I2 = {  father(eric, mario),  mother(alice, simon),
         mother(alice, mario), parent(eric, mario),
         parent(alice, simon), parent(alice, mario),
         anc(eric, mario),     anc(alice, simon),
         anc(alice, mario),    anc(simon, mario)}.

       Comment:- Is I2 an interpretation?  Rules 5, 6 & 7 are
       satisfied.  Also rules 3 and 4 are easily verified too.  With
```

```
some perseverance rules 1 and 2 are also satisfied by I2.
Note that the interpretation item anc(simon, mario) does not
contradict any rule or imply that parent(simon, mario) should
be present – which in fact it is not.  I2 is a model of the
above P.
```

```
Remark: interpretation three
I3  = I2 – { anc(simon, mario) }.

Remark:  I3 is a model of the above P.
```

Clearly, there can be many interpretations of a program that are a model. A useful result is that the intersection of two models is another model. A further observation (i.e. resulting from this result) is that there are models that on subtracting an element from them will not remain a model. For example by taking off any element from interpretation I3 disables its status of a model for the program. Models with this property are called *minimal models*. Furthermore if there is a model that is contained by all other models then it is called the *least model*. At this point one can easily prove an important result for positive Datalog programs: every program has a least model (one can follow the proof in [ULLMA90]).

6.1.5 Queries and Safe Answers

Having compiled a program (e.g. a deductive database) and its interpretation is a model then it is natural to expect an interaction with the model. Some of these interactions are called *queries*. A query allows the system to verify whether, or not, a clause is implied by the model of the program. In a *closed query* the response required is true or false. For example (and using Prolog's style for queries ?query) the query "? father(eric, mario)." would return true. In other queries, and where variables are introduced in the clause, the response includes all the facts that match the query. An example of an *open query* is "? father(X,Y)." that would return all matches for variables X and Y in the model. Note that variables in queries are implicitly universally quantified and their scope is the query clause.

Queries present us with an intriguing problem related to the finiteness of their answers. Can we have a query that returns an infinite response? The direct and correct answer is a yes. The use of built-in predicates can easily make a query answer infinite. Common *built-in* predicates over the infinite integer domain include equality of integers, the greater than, and the greater than or equal, and these are not usually represented with finite relations. Also representation is difficult as our predicate's arguments are not sorted (e.g. typed to an

integer) and therefore the domain to range has to be made clear. An example of a query that generates an infinite response is "? X >= 16." (i.e. find instances that are greater or equal to 16). On the other hand the following query "? age(peter, X), X >= 16." does not return an infinite response. Another subtle way to generate infinteness is to introduce a rule that has a free variable present in its head but not in its body. An example from Ullman [ULLMA90] is the rule: "loves(X,Y) <- lover(Y)." – all the world loves a lover. The relation loves is infinite if variable X ranges over an infinite world.

One solution is to disallow variables in built-in predicates or rule heads that are not anchored down in a program's predicates (i.e. an atomic wff). Queries and rules that have all their variables anchored (or limited) are *safe*. The standard practice for identifying safe rules is to use syntax-oriented constraints on a program's rules. These constraints are (as Ullman specifies [ULLMA90]):

> "1. Any variable that appears as an argument in a predicate of the body is limited;
>
> 2. Any variable X that appears in a sub-goal X=a or a=X, where a is a constant, is limited;
>
> 3. Variable X is limited if it appears in a sub-goal X=Y or Y=X, where Y is a variable already known to be limited."

6.1.6 Datalog Evaluation: Datalog to Algebra

The model-theoretic semantics provides for understanding the meaning of a Datalog program. This understanding is closely associated with the declarative meaning and consequently, from a processor point of view, we would like to have a mapping from declarative constructs to procedural constructs to enable an evaluation of the program. A likely target is the relational algebra. It is a straightforward algorithm to translate a simple sub-set of Datalog programs into algebraic expressions (whose input is an EDB) that evaluate the program.

Let us describe the sub-set of Datalog programs we can provide an algebraic evaluation for. The limitations are: positive Datalog programs; each variable appearing in a rule is limited; and no intentional predicate (i.e. rule) is recursive. For these types of programs our model theoretic semantics, proof theoretic semantics, and the algebraic computation evaluation

coincide. Indeed the Datalog programs in this category are simple (i.e. the example above is recursive and therefore not applicable to the following conversion).

The general outline of the evaluation algorithm depends on initially re-ordering the predicates in a program by their evaluation dependency; which is easily extracted by examining each predicates sub-goals. For example the evaluation of predicate **PARENT** (in the above example) depends on having evaluated its sub-goals; i.e. the predicates **MOTHER** and **FATHER**. The EDB predicates, like **MOTHER** and **FATHER**, do not depend on any other predicate. Some predicates do depend on themselves to be evaluated (i.e. recursive) with predicate **ANC** being such–this is a reason why we are proposing this evaluation method on non-recursive programs. Through this dependency relation one can build a graph – called a *dependency graph*. In a dependency graph each program-defined predicate is represented as a node, and each dependency relation is depicted through a directed edge. The dependency graph of the example program is presented in figure 6.1.

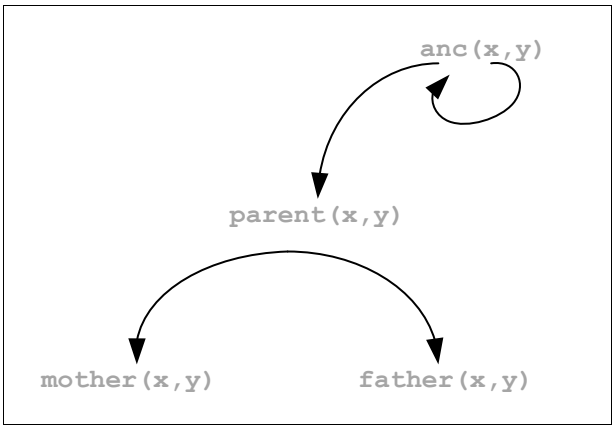


Figure 6.1 – Predicate dependency graph – negated sub-goals are adorned with a negation symbol

According to the predicate dependency graph of a programs one starts by identify rules that depend on predicates that have already been evaluated. For each of these one then needs to extract from the program all the rules in which this predicate is its head. And for each of these rules we need to form a relation that is the natural join of the sub-goal's relations (which from the dependency graph we know have already been evaluated). During this process one also needs to restrict the extent of the new relation through any constants or built-in predicates found in the sub-goals. The next step is to project attributes required by

the rule head from the natural join created. If the predicate being evaluated has a number of rules then one must union each evaluation. Some technical details are left out here; but the interested reader can refer to the whole procedure as found in Ullman's textbook [ULLMA90] (algorithm 3.1 p.109, V. I).

How to evaluate query parent(x,y)?

```
eval(parent(x,y)) == UNION(mother(x,y),father(x,y)).
```

Assume we have another rule to introduce the predicate grandparent(x,y), then rule and a query evaluation follow:

```
grandparent(x,y) <- parent(x,z), parent(z,y).
```

The evaluation of query grandparent(x,y) follow:

```
eval(grandparent(x,y)) ==
    PROJECT([1st,4th], SELECT([2nd=3rd],PRODUCT(PARENT,PARENT))).

Remark: 2nd and 3rd represent the second and third column
Remark:   of PRODUCT result
```

It is relatively easy to show that this algorithm produces the facts that can be proved from the database and that these facts (i.e. IDB and EDB) correspond to the unique minimal model.

6.1.6.1 Recursive Datalog Evaluation

We now need to consider an algorithm to compute the minimal model for a positive and recursive Datalog program. It is not hard to accept that the IDB rules are a basis for constructive build-up of the IDB relations. This observation, together with our previous evaluation algorithm for non-recursive programs form a framework of the required computation. Understandably the crucial point of this framework is that in recursive rules a predicate is mentioned both in the head and in the body. The technique used to solve recursive rules evaluation is the *fixpoint mapping*. In the fixpoint evaluation of a recursive rule we start with an empty relation, and execute an assignment from the EDB to the IDB and hence derive new facts for the recursive relation from new assignments until no new facts are derived through the recursive rule. This technique is usually referred to as the naïve evaluation and is attributed to Chang [CHANG88]. The pseudo code that follows is from [ULLMA90].

```
remark: let us assume that the datalog program has k EDB predicates
remark: and M IDB relations
FOR I: = 1 to M DO
    Pi := 0;        -- set predicate extent to null
REPEAT
```

```
    For I := 1 to M do
          Qi := Pi;
    For I := 1 to M do
          Pi := eval(pi, R1, …, Rk, Q1, … , Qm);
UNTIL Pi = Qi for all I, 1 <= I <= M;
OUTPUT Pi's
```

The fixed point of a program given an EDB and the derived IDB relations form a model of the program. It is a well-known result that Datalog programs have a unique minimal model and this coincides with their unique minimal fixed point. At this point an important bridge is our requirement that the naïve evaluation of a program relative to an EDB does reach a fixed point and that this fixed point coincides with the minimal model of the program.

To show that the naïve evaluation converges to a fixed point we need to establish that our call to the evaluation round is *monotonic* (the output is not smaller than the input) and that there is an upper limit of rounds. The relational algebraic operations of 'union', 'select', 'project' and 'product' are monotonic (but the 'diff' operator is *non-monotonic*). As the algebraic expression built by the evaluation round is based on these operands then the evaluation call per round is monotonic. To establish that there is indeed an upper limit to the number of rounds one can use the number of constants and predicates in the program to establish the limit's magnitude (i.e. each evaluation's increment fact's arguments are constants from the program and the number of predicates and arities are known). The final point of the proof is the establishment that the fixed point reached is indeed the least fixed point.

The naïve evaluation approach is greatly improved by the *semi naïve* evaluation which basically prunes the search space in each round to the newly derived facts (i.e. new facts generated in the previous round) – Bancilhon is credited with its introduction [BANCI85]. A totally different approach is the *magic set* approach [BANCI86] that combines bottom up and top down approaches by passing evaluation information sideways and rewriting the original program to reflect this. Also a control algorithm is required to drive the evaluation – should result be a set at a time or fact at a time.

6.1.7 Extending Datalog with Negation

It is a fact that certain rules (or queries) are not representable through Horn clauses. A particular class of these includes the use of negation on a predicate. Through negation a

rule can express inferences on exceptions and on differences (e.g. universal quantification). Strictly speaking a Horn clause structure does not allow negated predicates in its body. Datalog^{neg} is the language of Datalog programs that allows negation in rules.

A rule with a negated predicate that could be augmented to the above example is:

```
true_anc(X, Y)  <- anc(X,Y), not parent(X, Y).
```

An algebraic equivalent would be:

```
true_anc(X, Y) := diff( anc(X,Y), parent(X,Y)).
```

Another algebraic expression that evaluates the same rule is:

```
true_anx(X,Y) := product( anc(X, Y),
                  comp_parent(X,Y)).
Remark: - Where comp_parent(X,Y) is the compliment of relation parent
  with respect to the constants appearing in the discourse (i.e. U).
  Therefore comp_parent(X, Y) = diff ( product(U,U), parent(X, Y) ).
```

Although complementation seems innocuous it does create problems for our evaluation algorithm if the complemented relation has an infinite extent. Furthermore not all rules with negated sub-goals can use the complementation approach (note that predicates **ANC** and **PARENT** are union compatible and all variables are restricted). One of the first problems to consider is rules with variables instantiated only in negated sub-goals. An example of such a rule is:

```
Remark:  predicates male and married are EDB relations
bachelor(X) <- male(X), not married(X,Y).
```

The complementation of married is all pairs of objects that are not married to each other. Projecting the **MALES** from this complement one finds males who aren't married to anyone. A good fix is to re-write the rule as:

```
remark: re-writing bachelor
husband(X)  <- married(X, Y).
bachelor(X)  <- male(X), not husband(X).
```

An important observation is therefore to prohibit Datalog^{neg} programs rules with variables appearing only in negated sub-goals. Given the above re-writing technique for such rules this is not an impossible restriction to achieve.

Still not all is clear with Datalog^{neg}. It is a well-known result that such programs might have a number of minimum fixpoints but **no** least fixed point. The choice of which meaning is correct becomes critical. Over the years it has become customary to choose a minimum fixpoint derived from yet another Datalog program restriction – the Datalog^{neg} programs

with stratified negation. One has to quickly point out that the fixpoint chosen from a stratified Datalog^{neg} is not necessarily a least fixpoint.

What is a stratified Datalog^{neg} program? A program's rule is stratified whenever there is no path (in the program dependency graph) between the head and its negated sub-goal(s). Negated sub-goals in a dependency graph are represented as for any sub-goal but the edge is adorned with the negation sign. There are a number of algorithms to test for stratification, but some are more interesting as they could produce a number of rules strata. The stack of rule's strata is ordered in such a way that a rule with a negated sub-goal is in a higher startum then the sub-goal stratum. Consequently ascending with the order of the starta allows our rule's evaluation of interpret negated sub-gaols as if these are EDBs.

We are now in a position to give a framework for evaluating Datalog^{neg} programs. It is reasonable to assume we are evaluating a program whose rules are safe and stratifiable. We effectively use the strata to guide our evaluation of the program as each level is dependent on the current stratum and the lower ones – therefore the evaluation of each stratum through the naïve evaluation is possible. There is of course the requirement to evaluate a negated sub-goal. Toward this end we need to define a set of symbols, e.g. called DOM, to be the union of the symbols appearing in the EDB relations and the program. If p_i is the negated sub-goal (i.e. not p_i), we know that p_i has been evaluated in a lower stratum, and therefore the n-way product P_i (where n is the arity of P_i) less the tuples of P_i is the required compliment of p_i. This hinges on the fact that since the rules are safe; therefore no new constants can be generated and all variables appear in a non-negated sub-goal. The fixpoint reached through this evaluation is called the *perfect fixed point*.

6.1.8 Extending Datalog with Functions

There are a number of advantages to be had if our Datalog language is allowed to use functions to recursively define terms. For example Datalog^{func} allows us to identify objects through their components rather than with explicit constants. As a result our domain of objects need not be finite or fixed for each program prior to evaluation. Another use of functions is to build data structures (other than tuples)–an excellent example is the list.

The list data structure requires functions and recursive Datalog. Of course introducing functions has a bearing on the safety of Datalog programs.

The syntax of Datalog has to be extended to accept functions. In Datalog's wff with functions the terms are now defined as a constant, a variable, or a function (i.e. $f(t_1, \ldots, t_n)$ where t_i are terms and f is the function's name).

The semantics of Datalog has to be appended with an interpretation for functions. In $Datalog^{func}$ there is a mapping for each n-ary function from U^n to U.

The evaluation of a $Datalog^{func}$ program has a basis in the evlaution of Datalog presented earlier. It is interesting though, to show how simple $Datalog^{func}$ programs offer some difficulties. For example, the following program defines an infinite relation. This program's evaluation never ends, as the fixed point is never quite reached. Nonetheless we can heartily ascertain that this relation is well behaved in terms of the fixpoint evaluation. In fact the naïve evaluation could be "fixed" to allow the instantiation of a solution in finite time (e.g. is a tuple in the relation – i.e. "-? int(suc(suc(…(suc(0)) …)))").

```
            int(0).
            int( succ (X) ) <- int(X).
    Remark: Where succ is a function (from integer to integer) that maps
a
            successor to the argument supplied.
```

The basic technique used for upgrading the naïve evaluation is "term matching" that is required between a sub-goal's variables and ground atoms. This term matching is used to instantiate objects during the naïve evaluation rounds. In essence the naïve evaluation takes a $Datalog^{func}$ program whose rules are safe, and the EDB predicates. On invoking the algorithm, if the result is finite then the least fixed point is found. If the least fixed point is infinite, it produces an infinite sequence of approximations with the fixed point the limit.

6.1.9 Datalog and Relational Algebra

We have seen that the relational algebra is not able to express transitive closure–[AHOAL79]. On the other hand $Datalog^{neg}$ can do what the algebra does and more. Yet there are some queries that $Datalog^{neg}$ cannot do, but using $Datalog^{neg+func}$ shrinks further this class of queries.

6.2 Algebras as a Target for Declarative Languages

There has also been an interest, i.e. since relational calculus and algebra, in mapping declarative programs, e.g. Datalog programs, into algebraic constructs. A number of reasons exist: the first being that algebra can express the semantics of the Datalog program; and second is that after translating a program into algebraic expression, query rewriting and optimisation is possible. In the following section we describe the evolution of database algebras; furthermore chapters eleven through thirteen describe this research's effort to supplement declarative queries with algebraic ones.

6.2.1 Relational Algebra, Nested Relational Algebras & Object Algebras

An algebra is a well-known mathematical structure and it is defined through a pair. The pair consists of a set of objects and a set of operators. Every operator has its own set of algebraic properties; these include commutativity, associativity, and distributivity.

In most database retrieval algebras the operators take either one range of objects or two. Database algebras are procedural. The kinds of algebraic properties an operator has affects how it may be manipulated during query rewriting (which is essential in query optimisation).

Codd's relational algebra is considered to be the watershed for database languages [CODDE70, CODDE72]. The algebra has five operators, at minimum, works over a relational data model, and it is closed as it produces a relational table for each algebraic expression. The five basic operators are union, diff, product, project, and select. A number of other operands like intersect and division are a composition of the listed five. The language constructs possible from relational algebra are a sub-set of computational functions: therefore it is computationally incomplete. The algebra is equipollent to and convertible to relational calculus; which is a declarative language.

After Makinouchi's [MAKIN77] work on un-normalised relations other algebras, based on the nested model, started to appear in the mid-eighties. One cluster of papers appertains to Scheck and Scholl in [SCHEK85], [SCHEK86] and [SCHOO86]. The data model was nested relational (see figure 3.2 in chapter 3) and the operators supplement the relational ones with nest and unnest. The algebra was closed within the nested relational model. Over

this data model Thomas and Fisher [THOMA86] seminal paper introduced the algebraic properties of composing nest and unnests operands.

The group led by Roth and Korth (references [ROTHM87] and [ROTHM88]) used the nested relational model too. A good technique found in their operands is the recursive unravelling of nested tuples to retrieve the inner tuples of a nested relation; for example in union and in projection. Also their predicate in the select operand allows for set comparison. Roth *et al.* papers include a declarative language.

Pioneering database algebra for the complex object model came from Abiteboul and Beeri [ABITE95]. The data model uses the set and tuple for structure constructors without any restriction on their ordering. The database is value based and all sets are homogeneous. The algebra had the power-set operator and a replace operand (i.e. it applies a restructuring or mapping operation on an indicated range of values).

One of the first object algebras was by Shaw and Zdonik in [SHAWS90]. The data model included object identity, inheritance, and methods. The operators are influenced by the nested algebras but also an influence from functional programming is evident. The introduction of functional programming, which is very effective to restructure object structure, requires that some of the algebra's operators are higher order. The operators include: select, image, project, union, difference, flatten, duplicate elimination, coalesce duplicates at a level of nesting, nest, and unnest. These authors have been criticised for having higher-order features in first order systems; i.e. many of the algebras presented were not higher order [SUBIE98].

An early and thorough proposal was of Straube and Ozsu [STRAUB90]. In their work there is a calculus, algebra, and query processing. Their data model has basic object-oriented features but depends on higher-order operations.

An extensive object algebra is that from Leung *et al.* [LEUNG93] AQUA proposal; the researchers behind this proposal had already contributed to other algebras. Also AQUA anticipates supporting a wide array of data type constructors over their explicitly stated set and multi-set data type constructors. AQUA comes with twenty operations. For example there is a separated operator for join, outer join, and tuple join. Furthermore one can pass

a function as the join condition between two ranges of objects. It also includes an aggregation operation.

One of the neater object algebras is QAL by Savnik *et al.* [SAVNI99]. Again its origin is the relational and functional models. A novel introduction is that in the apply operator a path expression is an argument and is used as the query range rather than a class extent. This can eliminate a number of projects and unnesting operands from a query expression. Also the range of the operands can include the 'data dictionary' collection too. QAL operands include: union, difference, select, tuple, close, apply and 'applt at', nest and unnest, and group. A similar approach, based on a many sorted algebra, has also been developed by Lellahi and Zamulin in [LELLA01] and [ZAMUL02].

An interesting trend is that some of these algebras have more recently been re-packaged for use with XML databases. Buneman *et al.* [BUNEM96] paper in SIGMOD is a very early paper that set an early path to this area.

6.3 F-logic

Are there any object-oriented extensions to Datalog? Indeed there are such language extensions. An excellent example is Abiteboul and Kanellakis IQL [ABITE89] proposal which includes Datalog, negation, object creation, and inflationary fixpoint semantics (rather than the above perfect model semantics) as its basic characteristics. In general the connection between logic programming and the object-oriented model was through these support mechanisms: complex terms (i.e. in logic programming) to complex values (i.e. in object-orientation); predicates and their extensions to classes and their instances; object identifiers in predicate as labels to object identity; object-identifier-based sharing to object based sharing; and unification through term generalisation for inheritance. One problem is the "encoding" blurs the distinction between the paradigm's features and the implemented program.

Yet another approach is to couple (loosely or tightly) between a logical programming language and an object-oriented language. An excellent example of these genera is the proposal by Dinn *et al.* [DINNA95] (and later a working version) for ROCK & ROLL. Here the data model is based on a semantic data model. ROLL allows rule specification while

ROCK is a procedural language. This project is a success on a number of points: usage, implementation and extending ROCK & ROLL's capabilities (i.e. Active and Spatial additions exist). A criticism that one usually levels at this approach is how well does its extensions match – there could be an impedance mismatch occurrence between the "internal" components.

As we had already anticipated earlier in this section, there is yet another approach to defining deductive object-oriented databases whereby basic notions of object-orientation are entrenched (or canonised) in the underlying language. From the onset, one must state that a tangible difficulty is to include features (e.g. object-oriented themes and variations) but remain within the confines of first-order logic theory (i.e. technically the semantics are first order).

F-logic offers objects, object-identity, *ISA* relationships, methods (e.g. scalar or multi valued), non-monotonic inheritance (i.e. structural and behavioural), and typing. Schema and objects in F-logic are homogenous. The syntax is higher order; but the semantics are first order (in fact, for F-logic a model-theoretic semantics and a sound and complete proof theory exist). Another proposal along this path includes Orlog [JAMIL92]. The general problems of this approach are: remaining within first-order semantics; evaluation efficiency; and the complexity when using the language (e.g. by application programmers).

6.3.1 F-logic: the Language

F-logic language, $\mathcal{F}$-$\mathcal{L}$, alphabet comprises:

- The set of object constructors, $\mathcal{F}$ has the role of functions. In F-logic functions of arity 0 are constants, while functions of greater arity denote larger terms that have been built from simpler ones;
- An infinite set of variables, $\mathcal{V}$, (we use Prolog's practice of using uppercase to identify these more easily);
- A number of auxiliary symbols – e.g. (,), $\rightarrow$, $\twoheadrightarrow$, $\mapsto$, $\mapsto\!\!\!\!\!\!\rightarrow$, $\Rightarrow$, $\Rightarrow\!\!\!\!\!\!\rightarrow$, etc;
- The customary set of logical connectives and quantifiers, $\vee$, $\wedge$, $\neg$, $\leftarrow$, $\forall$, $\exists$.

An *id-term* is a term (as in Datalog) and is composed of functions and variables. The Herbrand Universe, $\mathcal{U}(\mathcal{F})$, is the set of all ground terms. These ground terms are the *logical object identifiers*.

In $\mathcal{F}$-$\mathcal{L}$ we have *molecules* (rather than atomic formula – i.e. a collection of facts) to build wff. The structure of molecules follows.

i) *An ISA assertion takes the form:*

```
Remark: object denoted by id-term b is a subclass of object
        denoted by id term a

       b :: a.

Remark: object denoted by id-term o is an instance of object
        denoted by id term a.

       o : a.
```

ii) *An object molecule for an object denoted by id-term O takes the form:*

```
Remark: 1: the object will respond to methods available in the list
Remark: 2: there is no implied order in the list

       o[ a semi-colon delimited list of method expressions ].
```

iii) *A number of method expressions are possible.*

A *scalar* (i.e. single headed arrows) and *set* (i.e. double headed arrows) *non-inheritable data expressions* are represented as:

```
Remark: 1: scalar_method, set_method, Ai, R, Si are id-terms
Remark: 2: the Ai are the method's arguments and R is the result
           for scalar methods, while the result of a set method
           are Si id-terms

       scalar_Method@(A1, …, An) -> R  where n >= 0
       set_Method@(A1, …. , An) ->> { S1, … , Sm}   where n >= 0, m >= 0
```

An *inheritable method* allows F-logic to pass a method through the ISA relationship (i.e. :: and :). An inheritable data expression are represented as (i.e. add the star to the single and set symbols):

```
scalar_Method@(A1, …, An) *-> R where n >= 0
set_Method@(A1, … , An) *->> { S1, … , Sm } where n >= 0 and m>=0
```

A scalar and set-valued signature expression is represented as:

```
       Remark: 1: the id-terms Ai denote the type of the arguments
       Remark: 2: the id-terms of Ti denote the type of the result,
                  and if m > 1 then the return object has to have
                  all the listed types

scalar_Method@(A1, …, An) => ( T1, … , Tm )  where n >= 0 and m >= 0
set_Method@(A1, … , An) =>> ( T1, … , Tm )  where n >=0 and m >= 0
```

After considering the basic structures in an F-molecule it is worth emphasising some other points on how these are built. An object's attributes are given through data expressions: i.e. details on an object denoted by id-term "peter" could be given as:

```
peter [ name -> "PETER" ; sex -> male ; work -> manager ; tels ->> {25, 21}].
```

The signature of persons, ISA and instance-of assertions could be given as:

```
person [ name => string ; sex => sextype ; work => jobtype ; tels =>> int ].
peter : employee.
employee :: person.
```

For the data type constraints to be satisfied (we shall see later what we actually mean here) we need to add the following instance-of facts:

```
male:sextype. female:sextype.
manager:jobtype. other:jobtype.
```

If we change the signature molecule with the following and re-evaluate:

```
person [ name => string ; status *-> employed ].
```

Then the following query will confirm that "peter" is still an employee.

```
?- peter [ status -> employed ].
```

If we fire the following query notice how F-logic responds (note two solutions are offered rather than one for a set):

```
?- peter [ tels ->> N ].
N / 2503
N / 2131
```

If "peter's" details are changed (and the program is then re-evaluated) and then we fire the same query it returns false. This is because the methods defined for an object over-ride the inherited ones. Also to note is that through the instance relationship the type of molecule changes from an inheritable to a non-inheritable type.

```
peter [ name -> "PETER" ; status -> unengaged ].
```

F-molecules are allowed to have a nested structure. For example:

```
peter [ addr -> petehome [ str -> "main"; city -> "cosmo" ] ].
```

The following query would bind "cosmo" to variable Y:

```
? peter [ addr -> X [ city -> Y ] ].
```

Clearly this technique goes some way towards implementing complex objects. (In a later sub-section, where F-logic predicates are introduced, one can combine F-molecules and predicates to represent more naturally a wide variety of complex objects).

An interesting example is the following wff data definition (i.e. with id-term "atr" repeated):

```
o[atr -> a1; atr ->> {a2,b2}; atr *-> a3].
```

How are rules specified? The following rule derives a new attribute for any person who shares a phone with another person (e.g. including herself):

```
P[shares@(T)->>P2] <- P:person , P[tels->> T] , P2:person , P2[tels->>T].
```

The commas are used for denoting conjunctions between F-molecules. It is also appropriate to note that id-terms denote either an object (an entity instance) or a method. The context is determined by the placing in the wff and the type of arrow.

F-logic's wff (or F-formulas) are built in terms of other F-formulas. F-formulas are therefore generated iteratively from: molecular data expressions are F-formulas; F-formulas using the logical connectives; and F-formulas built from quantifiers (e.g. if F is a F-formula and X a variable then forall X F). A literal is an F-formula or a negated F-formula.

6.3.2 F-logic's Semantics

The standard reference on F-logic [KIFER95] explains the semantic structures through a model-theoretic approach. In the same reference these are called F-structures. An interpretation for F-logic language is a ten-tuple. The first attribute is the domain of the interpretation. The second is an irreflexive partial order on U that captures the subclass relationship (i.e. ::). The third is the binary relationship to model the instance-of relationship (i.e. :). There is a relationship between these structures as it enforces the deep extent relationship (i.e. if a is an instance of b and b ISA c then a is an instance of c). It is important to mention that there are no other restrictions on the instances of the ISA structures. This allows, for example, an object to be an instance of itself (e.g. useful in AI for depicting the typical object). A fundamental point to understand with F-logic is that in the "a is an instance of b" assertion b is not a subset of U but b is an element in U denoting a set. The claim that F-logic has a first-order semantics is thus satisfied. The forth attribute interprets each n-ary object constructor (i.e. functions) – just like the first-order predicate calculus [ENDER72]. The other six attributes of the interpretation of F-logic deal with method expressions (i.e. the different arrows).

While the deep extent relationship is entrenched in the interpretation; the relationship between a method and its signature is not and it is defined elsewhere (i.e. at a meta level through type correctness rules). It suffices to note, at this point, that methods have a

functional form and the structures for scalar and set method signatures map with the properties of Cardelli's semantics of multiple inheritance [CARDE88].

As in predicate calculus variable assignment is straightforward. It is defined as a mapping v from the set of variables $\mathcal{V}$ to the domain U. This extends to id-terms.

How are F-molecules satisfied? A molecule T[…] is true under an interpretation I given a variable assignment v if object $v(T)$ in I has the properties of molecule T[…]. This is succinctly represented as: $I \models v$ T […]. Of course this notion is specialised for each type of F-molecule. Also the meaning of F-formulas with logical connectives, for example φ AND ψ, is defined as $I \models v$ φ AND ψ if and only if $I \models v$ φ and $I \models v$ ψ. The meaning of F-formulas with quantifiers is, $I \models v \forall X$ φ iff $I \models u$ φ for every u that matches v (except possibly on X). An interpretation I of F-logic is a model of a closed formula φ if and only if $I \models$ φ.

6.3.3 F-logic and Predicates

F-logic can simulate Datalog. One technique reported in the reference paper is to encode an n-ary predicate p as instances of a class p. Specifically:

```
p(T1, … , Tn)[arg1 -> T1; … ; argn -> Tn] and p(T1, … , Tn):p.
```

The report also shows how predicates could form part of the interpretation of F-Logc programs. Basically we set to introduce a set of predicates, P, and a meaning for these in the language interpretation (i.e. the eleventh entry). With this introduction Datalog and logic programming (as presented earlier) become a sub-set of F-logic.

6.3.4 F-logic's Proof Theory

It is important to state that F-logic has a sound and complete proof theory for logical entailment of F-molecules (the statisfiability requirement defined earlier). The theory has a high number of inference rules – twelve and an axiom in all. Logic's programming resolution, factoring and paramodulation are present here too. The higher number of inference rules is due to the number of features entrenched in the semantics of the language.

6.3.5 Logic Programming with F-logic

In a previous example we have shown how F-logic can use rules to derive the value of an object's attributes (e.g. share@(tels)->>integer). It is evident then that Horn clauses are allowed in F-molecules. If an F-logic program's rules are Horn clauses then the model

intersection property holds (i.e. the least model). If the rules are recursive then the evaluation technique shown for Datalog (the use of the fixpoint operator) is also applicable here and gives the same result.

If in our F-logic methods their derivation require general programs (e.g. use of negation) then the coincidences between fixpoints, minimal models and satisfiability is lost. If a program's stratification is reached then the program's evaluation is a perfect model.

Whereas in Datalog it was possible and acceptable to assume two objects are equal if they were assigned the same constant it is not possible to entertain this approach in F-logic. The reason being that we assign functions for id-term and also the semantics of the language will interpret that two object's id-term must be representing the same object. For example if we have two ISA assertions like person :: human and human :: person then person and human are equal. This is because of the class hierarchy acyclicity semantics. One suggestion from Kifer *et al.* is to split the program into two. The first part will allow inferences about equality (as in the example of person :: human), while the other part will prohibit inferences on equality unless it is specified explicitly (id-term of person and of human are denoting the same object).

One last point in this section is about queries. In F-logic a query as the form of "?- Q", where Q is an F-molecule. As is common to logic programming the answer to the query, with respect to an evaluated F-program, is the smallest set of F-molecules that are found in the program's model. There is a second condition in F-logic that needs attention. Let us re-examine a query and its response we had written about:

```
?- peter [ tels ->> N ].
N / 2503
N / 2131
```

If we syntactly change the query into the following we do not get "true":

```
?- peter [ tels ->> { 2503, 2131 } ].
```

To solve this we must check our model for closure (i.e. |=) with respect to satisfiability. Therefore our queries must be closed with respect to |= .

6.3.6 F-logic and Typing

In F-logic the data expressions (i.e. methods) are functional and there is a possibility of passing arguments and returning either an object or a set of objects. It comes as no

surprise that arity polymorphism is allowed in F-molecules. An example of arity polymorphism is the method salary in the following F-molecule:

```
person [ name => string; sal@(1995) -> 500; sal@(1995, manager)-> 350 ].
```

We have also seen that an F-logic program can specify a method's signature; but we commented (in the semantics section 6.3.2) that the typing enforcement is not entrenched in the semantics. To rectify this F-logic provides "well-typing conditions" at a meta-level.

The first condition is dedicated to non-inheritable methods and states that every such method has to be covered by a signature found in an object (whose context would be of a class). If we have the molecule a[m $\rightarrow$ r] and an assertion a:c then there must be a signature in c that covers m (e.g. c[m $\Rightarrow$ t]). The second deals with inheritable methods (i.e. $\bullet\!\!\rightarrow$) and its mode is different from the first. If we have a molecule c1[m $\bullet\!\!\rightarrow$ t] and assertions c1::c and a:c1 then the signature that covers method m in object a is found in c1. A general F-logic program interpretation is typed if these two conditions are satisfied.

In general this result is not strong enough for some F-logic programs (i.e. a sub-set of general F-logic programs) as static type checking is not possible. For example if in a general F-logic program there is a rule that can generate new sub-class relationships then static type checking is not possible. Another occurrence of typing errors comes from rules that never fire (e.g. type errors in the body of the rule). Although this is lethal from some aspects (Kifer and Wu state that it is un-decidable to type check a general F-logic program [KIFER90]) there are some countermeasure possibilities. For example after program evaluation additional rules can then do the required static checks (we can use stratification techniques toward this end). Clearly some way to bridge the gap between logic programming and programming languages in terms of type checking is required (this line of research is attributed to Mishra's [MISHR84] work on introducing type inference in Prolog). We can even take some advantage of this situation. First the typing rules are written by us and therefore can address a range of typing formalism. And secondly we can focus type checking onto a sub-set of the program's interpretation (e.g. type-check those methods that are dependent on type matching to external object definitions).

We typically need to make two checks for establishing type correctness. One is *type safety* that checks that there is no method without a covering signature in the interpretation. The other is *type correctness* that checks every signature covering a method is satisfied in the interpretation. Remember if there are a number of result types specified in a method signature (i.e. m ⇒ {A,B}) then the resulting object must be in the extent of each type. A set of rules that check simple methods (scalar and without arguments) follow:

```
remark: assume a general f-logic program follows

…
type_safe_sm(O, M, R)  <- O[M -> R] , O:C , C[M=>( )}.
type_unsafe_sm(O,M,R) <- O[M-> R] , not type_safe_sm(O, M, R).
type_incorrect_sm(O, M, R) <- O[M -> R] , O:C , C[M=>D] , not R:D.
remark:  after the final re-evaluation fire the following queries
?- type_unsafe_sm(O,M,R).
?- type_incorrect_sm(O,M,R).
```

The typing modality in F-logic is therefore to apply typing rules on a meaningful program (i.e. but untyped) and then check for safety and correctness. The general approach is to consider the F-Program's signatures, introduce type inference rules (for un-typed methods) and then do type checking by evaluation. Effectively write rules and queries to type check.

6.3.7 F-logic and Inheritance

F-logic supports both structural and behavioural inheritance through the constructs :: , ⥅, and ⥇ respectively. Furthermore F-logic semantics has structural inheritance and its proof theory is sound and complete for the same semantics. The problematic part is the behavioural inheritance and its modality (e.g. overriding and multiple inheritance).

A good example of overriding (from [KIFER95]) is describing clyde (an elephant!). Consider the following F-Program:

```
royalElephant::elephant.

cylde:royalElephant.

elephant[ colour *-> "grey" ; group *-> mammal ].

royalElephant[ colour *-> "white"].
```

Since clyde is a member of the royalElephant class it inherits the inheritable method (i.e. colour •→ "grey") as a data method (i.e. colour → "grey"), and does not take the inheritance from elephant (i.e. colour → "white" is overridden). Also to note is that sub-class royalElephant does not take the colour inheritable method, but does inherit the group as an inheritable method (i.e. royalElephant [group •→ mammal] is true).

A classic example of multiple inheritance (also included in Kifer *et al.* reference [KIFER95]) is Nixon's Diamond. Consider the following F-program:

```
nixon:quaker.

nixon:republican.

quaker[ policy *-> pacifist ].

republican[ policy *-> hawk ].
```

Having an association with both quakers and republicans our Nixon is spoilt to choose between either inheritable method (i.e. policy). But note that once Nixon accepts a policy then he excludes the other policy (i.e. either pacifist or hawk). Therefore the instantiation of the query "?- nixon[policy→ X]" is determined by which inheritance has been taken first.

In both cases our concern that logical satisfiability becomes non-monotonic is a reality. The F-logic language takes a *non-deterministic approach* (with the choice criteria being fair). The idea is to choose one of the possible worlds since each possibility is really a canonical model. This mode of inheritance is known as the "credulous" variety. Other regimes, and sometimes less ambitious, have also been introduced by Kifer and others.

To integrate inheritance into a deductive object-oriented database, as presented through F-logic, a number of issues have to be understood and controlled. For example, how are inheritance and logical deduction to interact? Furthermore how are set valued data methods or ISA assertions to react in the presence of inheritance? The difficulty of these issues stem from an assertion derived through an inheritance that causes a chain-reaction of other deductions. The scope of inheritance (and its effects) is both the source and the recipient of the inheritance. Consider the following simple program:

```
a:b.
```

```
b[attr *->> c].

b[attr *->> d] <- a[attr ->> c].
```

Through inheritance a[attr ↠ c] is derived. Consequently the rule deduces that b[attr ⇸ d] is true. If we re-write object b's data molecule attr we get: attr ⇸ {c,d}. The question is should we re-inherit in object a from b? F-logic does not undo the first inheritance, but does not enable the inheritance of derived set-based inheritable methods. The justification for this decision seems to be twofold: reduce the amount of backtracking during evaluation and secondly it yields semantic simplicity (albeit procedural). The formal method proven by the authors is based on two steps: in the first we fire the inheritance occurrence from source to recipient objects and disable the inheritable method; while in the second phase we get the state reached through inheritance into a model. The inheritance firing, or triggering, stops when there are no more inheritable methods in the original program. An important point to emphasise is that inheritance triggering starts after the evaluation finds a model for the F-program less the inheritable methods. In practice the Nixon diamond example becomes amenable in this scenario.

It is an interesting exercise to consider how F-logic can mimic the inheritance modality of other systems. One example would be to model user-controlled inheritance. In this scenario the user (rather than F-logic) wants to decide which inheritance to allow triggering in the circumstance of multiple inheritance conflicts. A case in point is Eiffel's explicit multiple inheritance conflict resolution. Let us assume there is an inheritable method, it is scalar and includes an argument, and it is attached to a number of classes (these are c_1, … , c_n and say variable Class ranges over these), then we need a rule that parameterises the method for each of these classes.

```
Class [ method(Class) @ X *-> Y ] <- Class [ method @ X *-> Y ].
```

If we have an object (e.g. o) that inherits from c_1, … , c_n then this object will have inherited all the definitions of "method" – but each is different because of the "Class" parameter. Consequently if the programmer wants to invoke the method attached to class identified by c3 will need to write a query so:

```
?- o [ method(c3) @ X -> Y ].
```

6.3.8 F-logic Implementation: Flora-2

Flora-2 [YANGG08] is a deductive and object-oriented database (DOOD) development framework. It is based on F-logic, Hi-Log [CHENW89] and Transaction Logic [BONNE94] and is coded in XSB Prolog. Hi-Log and Transaction Logic features are available for meta programming and knowledge-base updates respectively.

Flora-2 implements most of F-logic but has some syntactic changes and a number of additions to it. The most obvious syntactic difference is that whereas in F-logic there is a difference in notation between scalar and set methods these have been generalised into set methods with cardinality constraints. Also some symbols like ";" and "@" have been replaced to align better with the underlying Prolog system. Path expressions are part of F-molecules and F-formulas. Additional facilities to F-logic found in Flora-2 include aggregates, for example count instances that satisfy a query, and more meta programming facilities that are very useful in database environments. A useful feature Flora-2 has is the generation of id-terms on demand.

It is also possible to configure the Flora-2 inference engine prior to an evaluation of a program. For example, one has a choice of options for equality maintenance and inheritance semantics.

One can install Flora-2 on Linux and Windows, and both XSB Prolog and Flora-2 are open-source. Plug-ins for Eclipse IDE and EMACS also exist. XSB Prolog, which is accessible from Flora-2 programs, has access to data managed by a DBMS through data connectivity facilities and protocols.

6.3.9 Flora-2 Session

The following is a sequence of interactions with a deductive database within a Flora-2 environment.

```
Remark Identify classes as an instance of 'class'
person      :class.
course      :class.
dept        :class.
lecturer    :class.

Remark ISA facts
student     ::person.
lecturer    ::person.

Remark Instances of person, lecturer and student
barbara     :person.
betty       :person.
mark        :lecturer.
mary        :student.
```

```
michael     :student.

Remark Creating an enumerated type
sex         [enum-> {'m','f'}].
unitgrade   [enum-> {'a', 'b', 'c', 'd', 'f', 'abs'}].
assessment [enum->{'assign', 'test', 'exam', 'interview'}].

Remark Adding attributes to objects
barbara  [fname->'barbara', telno->{t123}].
betty    [fname->'betty',   telno->{t456},  workson->{proj1,proj2,proj21}].
mark     [fname->'mark',    telno->{t5,t9}, degrees->{'b','d'},
          worksat->eeng,
          teach(2000)->intprog, teach(2001)->intprog,teach(2002)->intprog,
          teach(2001)->javaprog,
          coord->{intprog,softeng},
          prolead->{proj3, proj11, proj21},
          workson->{proj3,proj11}].
michael  [fname->'michael',  telno->{t9012,t3456},
          enrolon->bsc_comp, stage->'f'].
michael  [result('c')->dsa, result('c')->softeng, result('c')->pdb,
          result('c')->intprog].

Remark Declaring object signatures
       (including Flora-2 cardinality constraints)
lecturer [degrees*=>string,  worksat{1:1}*=>dept,       coord{1:3}*=>unit,
          prolead{0:*}* => project, teach(integer){0:*}*=>unit].
person   [fname*=>string,           telno{0:*}*=>telephone,
          workson{0:*}*=>project].
student  [enrolon{1:1}*=>course,    stage*=>string,
          result(string)*=>unit].

Remark Instances of 'class'are?
?- ?Any:class.
?Any = course
?Any = person
…

Remark: Deep extent query
?- michael:?C.
?C = person
?C = student

Remark: Deep extent query
?-?Who:person.
?Who = barbara
?Who = mark
…

Remark: A queries with an and (,)
?- mary[result(a)->?_U1 ,result(f)->?_U2].
No.

Remark: A query with an or (;)
?- mary[result(a)->?_U1;result(f)->?_U2].
Yes.

Remark: Path expressions – who got an A in a 2 credit unit?
?- ?X.result('a').credits=2.
?X = susan

Remark: To who & what type of assessment was awarded an A in a 2 credit unit
?- ?X.result('a').credits=2, ?X.result(?_G).ass=?T.
?X = susan, ?T = assignment
?X = susan, ?T = exam

Remark Constraint Denial - Uniqueness
?- ?P1.fname = ?P2.fname, not ?P1:=:?P2.

Remark Functional dependence
?- ?I1:unit[credits->?V,assessment->?A1],
   ?I2:unit[credits->?V,assessment->?A2],
    \+ ?I1 :=: ?I2, not ?A1 :=: ?A2.

Remark Who & what type of assessments does one take if got an A in a 2
credit unit
?- ?X.result('a').credits=2, ?X.result(?_G).ass=?T.
?X = susan, ?T = assignment
?X = susan, ?T = exam
```

```
Remark Invoke XSB Prolog from Flora-2 using call
?- call(sort([a,s,d],?S)@_prolog).
?S = [a, d, s].

Remark Aggregate query – degrees per person
?- ?P:person,?C = count{?P|?P[degrees ->?C]}.
?P = linda, ?C = 0
?P = lisa, ?C = 3
…

Remark Aggregate query – years during which a unit was taught
?- ?C = collectset{?Y[?U]|?U[taughtby(?Y)->?L]}.
?C = [2000, 2001, 2002], ?U = ddb, …
?C = [2000, 2001, 2002], ?U = dsa, …
…
```

6.4 Summary

In this chapter we surveyed the research area of deductive and deductive object-oriented databases. For many years Datalog has been used as a yardstick and model to study interesting problems related to databases, query processing and recursive query processing. An interesting feature of this research is the use of algebraic languages to evaluate, and optimise, Datalog programs.

An notable occurrence is that there has been, in the last three years, a resurgence of research in Datalog. Furthermore industrial applications are adopting Datalog based systems [HUANG11] in the following areas: declarative networking, data integration, and program verification.

Here have read about F-logic–a logic that is object-oriented friendly. Technically it has a first order semantics, has a complete proof theory, resolution-based proof procedure, and higher-order syntax. Flora-2, an F-logic implementation, appeared later. Interestingly F-logic has gained popularity not only in database theory and deductive databases but also in for example the semantic web. F-logic is an ideal target language for data-modelling languages and to execute declarative queries against an F-logic program (i.e. rules and facts). Given the current hardware trends, like very high computer processing throughput and large arrays of main memory, one can propose F-logic as an exciting option as it offers a strong basis for a deductive database.

Unfortunately this option has shortfalls. For example sharing is a problem (but it is not of interest in this research). Another problem is data size; inevitable there are databases whose size is greater than main memory available and therefore secondary storage is in need. Consequently F-logic queries need optimisation and it is in our interest to find

evaluation methods, at least for a subset of F-logic queries, that can be optimised much like relational tuple calculus is converted to relational algebra and then optimised. We present a procedural execution program for a subset of F-logic queries (or ODMG OQL queries) defined on an ODMG ODL data model that is optimisable and for which optimisation is more sensitive to the computational platform available at runtime.

Chapter 7

Integrity Constraints

7 Integrity Constraints

Integrity constraints are rules that any instance of a database structure must uphold.

Therefore, integrity constraints are properties of the schema. Also integrity constraints supplement a number of other database related aspects. For example state changing operations over a database have to adhere to and respect these rules.

Here are some common examples of integrity constraints. The first is the basic *primary key constraint* on a database structure. In a value based model the constraint requires that the concatenation of a number of attributes' value is unique for any set of instances of the structure. A second example is the *check constraint* which deals with what values a structure's attribute can take. A case in point is restricting the grade of a student to one of the following values (from 0 to 5). In databases explicit relationships between instances are ubiquitous and our third example is *referential constraint* (or *inclusion dependency*); each employee instance is related to a department instance. Our forth example deals with expressing the "*business rule*" that there can be at most four students with a grade level *5* classification. Finally our fifth example associates a constraint to a database operation in the following manner; on changing a student's details their name value must not change – this is an example of *transitional constraints*.

Once we have a representation of a database and its integrity constraints then we are in a position to deduce whether a database state violates any integrity constraint

From a pedagogical perspective it is useful to give a coarse classification of integrity constraints in data modelling (see figure 7.1). The first categorisation is whether the constraint deals with rules that the structure's instances must obey or whether the constraint is applicable when a database operation is applied in due course and restricted to the instances affected by the operation. The former are the *static constraints* while the latter are the *transitional constraints* (fifth example above). The operations of interest in the transitional case are either the *insert*, or the *update*, or the *delete*, or any permutation of the previous three.

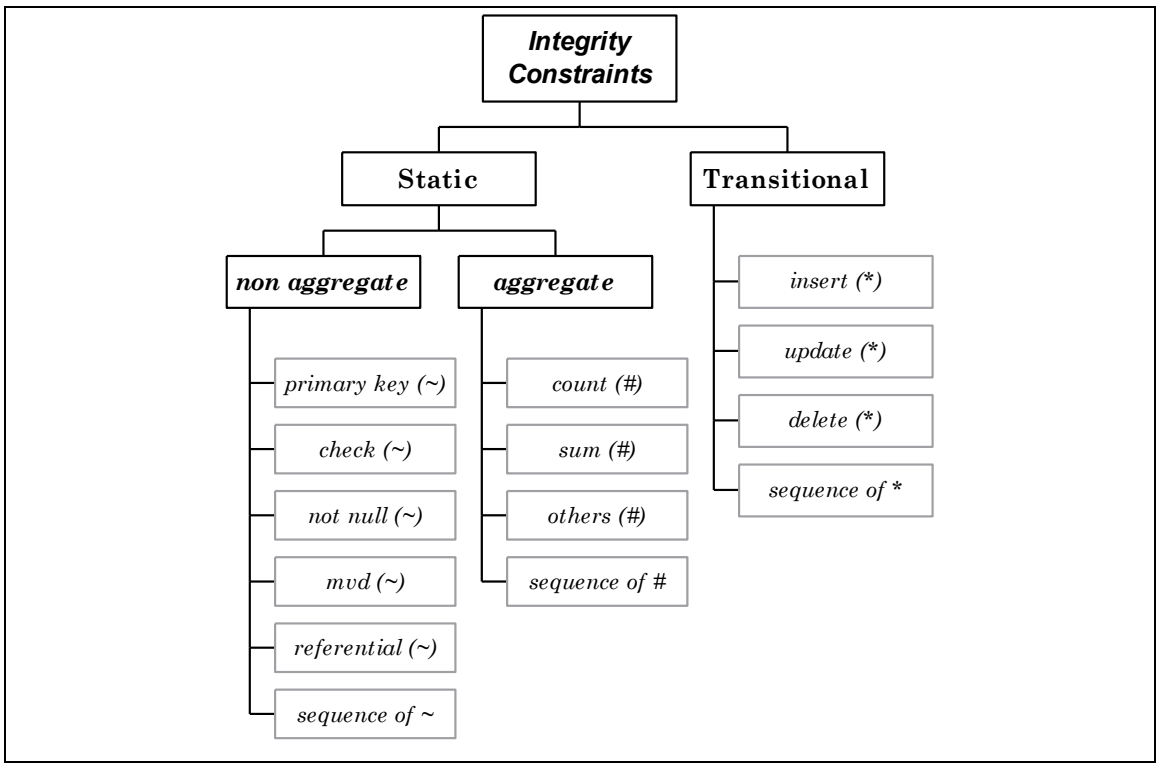


Figure 7.1: An Integrity Constraints Classification for Data Modelling

The static constraints are neatly divisible in two branches if we classify the integrity rules by whether the test is based on an instance value, or whether the test is applicable to an aggregate function on a collection of instances. The former are *the non-aggregate constraints* while the latter are the *aggregate constraints*.

Specifically the fourth example above is an aggregate constraint as it is based on comparing a value against the count of first class students from a set of instances (i.e. all students).

There is a good variety of non-aggregate static constraints of which one finds the primary key, the check, and the referential constraints (these are the first, second and third examples given above).

7.1 Integrity Constraints Representation

The leverage of using data-modelling constructs that are representable with a logical language is seen through the capabilities of describing and inferring on data, relationships, queries and constraints. Within this we are going to give a representation of integrity constraints through predicate logic closed formula.

The non-aggregate static integrity constraints are the easier to give a formal representation of, and also a basis for a presentation on the advantages of being able to depict and reason

about databases and constraints. Before we present the logical forms of these constraints we first need to present some notation and definitions. Let us assume our database is a collection of structures and these are relations – generally depicted as *rel()*. Each relation is made up of a number of attributes (identified as an offset in the structure or as A_i) to which a domain is associated with $dom(A_i)$. Also with each relation we associate a collection of integrity constraints applicable to the relation's instances. A relation's declaration and property, an *instance's definition* and a *set of instances* are presented as:

$$\forall x_1 \ldots \forall x_n \left(rel\left(x_1, \ldots, x_n\right) \rightarrow x_1 \in dom\left(A_1\right) \wedge \ldots \wedge x_n \in dom(A_n) \right).$$

$$\text{instance\_of\_rel} \in dom\left(A_1\right) \times \ldots \times dom(A_n).$$

$$\text{instance\_set\_of\_rel} \subseteq dom\left(A_1\right) \times \ldots \times dom(A_n).$$

7.1.1 Check Constraint

Relation **STUDENT** is made up of three attributes: **NAME**, **CLASS** and **DEGREE** subject. If we want to "ensure that each student has one classification from a list of values" then we would need to introduce a check constraint on the **CLASS** attribute (i.e. the second attribute in **STUDENT**). To represent this constraint we specify:

$$\forall x \forall y \forall x \left(\begin{array}{l} \text{student}(x, y, z) \\ \rightarrow y = 0 \vee y = 1 \vee \ldots \vee y = 5 \end{array} \right).$$

7.1.2 Primary Key Constraint

If we take the same relation and we need to specify that "**NAME** (in the first attribute in **STUDENT**) is the primary key attribute" then we specify:

$$\forall x \forall y \forall z \forall x' \forall y' \forall z' \left(\begin{array}{l} \text{student}\left(x, y, z\right) \wedge \text{student}\left(x', y', z'\right) \wedge \\ x = x' \\ \rightarrow y = y' \wedge z = z' \end{array} \right).$$

7.1.3 Not Null Constraint

To enforce the rule that "all students must be assigned to a degree course" (the third attribute of **STUDENT** is **DEGREE**) we use the not null constraint. This is specified with the rule:

$$\forall x \forall y \forall z \left(\text{student}\left(x, y, z\right) \rightarrow \neg isnull\left(z\right) \right).$$

This representation requires clarification in that checking if an argument of a predicate is null requires delving into and requiring certain implementation details.

7.1.4 Referential Constraint

If we would want to enforce the rule that "each student must enroll with at least one department" (the relation **DEPT_STUD_LIST** is made up of attributes **DEPARTMENT** name and **STUDENT** name) then we specify:

$$\forall x \forall y \forall z \left(\text{student}(x, y, z) \rightarrow \exists d \, \text{dept\_stud\_list}(d, x) \right).$$

7.1.5 Functional Dependency and Multi-Valued Dependency

The last type of our generic static constraint is the multi-valued dependency and this, as already mentioned, subsumes functional dependency. Actually a primary key constraint is a form of functional dependency – i.e. the key set determines all the relation's attributes.

An example of a functional dependency on the relation student is "for each student name and degree then there could be only one degree classification". The specification of a functional dependency is:

$$\forall x \forall y \forall z \forall x' \forall y' \forall z' \begin{pmatrix} \text{student}(x, y, z) \wedge \text{student}(x', y', z') \wedge x = x' \wedge z = z' \\ \rightarrow y = y' \end{pmatrix}.$$

For a presentation of the more general multi-valued dependency it is best to consider a few example structures. For example, if we want to represent the fact that a student has a number of contact points and a number of certificates (both contact points and certificates in possession are mutually independent). To represent this in a relation, called **STUD_C_H**, we opt for a predicate with three attributes named **NAME**, **CONTACT** and **DIPLOMA**. For consistency we have to include a tuple for each combination of contact points and certificates that are held by each student (i.e. introduce redundancy). An integrity constraint specification to support this is:

$$\forall x \forall y \forall z \forall x' \forall y' \forall z' \begin{pmatrix} \text{stud\_c\_d}(x, y, z) \wedge \text{stud\_c\_d}(x', y', z') \wedge x = x' \\ \rightarrow \exists x_2 \exists y_2 \exists z_2 \exists x_3 \exists y_3 \exists z_3 \begin{pmatrix} \text{stud\_c\_d}(x_2, y_2, z_2) \wedge \\ \text{stud\_c\_d}(x_3, y_3, z_3) \wedge \\ (x' = x_2 \wedge x_2 = x_3) \wedge \\ (y = y_2 \wedge y' = y_3) \wedge \\ (z' = z_2 \wedge z = z_3) \end{pmatrix} \end{pmatrix}.$$

7.1.6 Aggregate Constraint

To give a representation of a static aggregate constraint we can use the student relation. A specification of the forth example is:

$$\forall y \forall t \left(\text{count} \left(\exists x \exists z \, \text{student}(x, 5, z), t \right) \rightarrow t \leq 4 \right).$$

This requires some explaining. First we need a digression about predicate logic with function symbols and the satisfiability of this system. It is a well-known result that in such a case the system is not logically satisfiable. Therefore our representation avoids using any function symbols. In fact the method of representation above uses the symbol **COUNT** as an *aggregate predicate* with the following conditions and meaning (see [DASSK90] for a full discussion and formal justification). Each literal is composable either from first order literals or from aggregate predicates (e.g. the predicate count, sum, min and max are the typical examples mentioned). The meaning of **COUNT(F,*x*)** is the number of different true responses to the query **q** ← **F** in a normal database returning the number ***x*** in their response. An aggregate constraint is a closed aggregate formula.

7.1.7 Transitional Constraint

Representing transitional constraints also requires a technique to enable harmonisation with static and aggregate constraints. The introduction or purging of a fact in a database, and especially with deductive databases, causes a number of updates coded as a sequence of deletions and insertions. One technique is based on the pioneering paper of Nicolas and Yazdanian [NICOL78], entails the introduction of action predicates adorned with the transitional constraints. For example to implement the constraint that a student's name cannot change on an update operation we introduce an action predicate called **UPD_STUDENT** that has six attributes, namely the before and after values of student's original attributes. This update transitional constraint is specifiable as:

$$\forall x \forall y \forall z \forall x_1 \forall y_1 \forall z_1 \left(\text{upd\_student}(x, y, z, x_1, y_1, z_1) \rightarrow x = x_1 \right).$$

Our direct interest in integrity constraints for data modelling focuses more on static, than transitional constraints as the latter deal more with transaction processing.

7.2 Integrity Constraints and Databases

The previous examples, although motivated by data modelling, are only a small aspect of how integrity constraints and databases intertwine. In this section we intend to specialise this facet further. The main points are:

i. the huge number of instances relative to the number of constraints make executing database operations a computationally expensive activity;
ii. the data modelling of relationships are typically more complex than bare referential constraints shown previously;
iii. the conversion of integrity constraints into database queries of denial;
iv. the scope of an integrity constraint (e.g. is it an attribute, a relation or a schema constraint) and the right place to attach the constraint to.

7.2.1 Efficiency

In a deductive database we expect to have a good number of rules and for some of the predicates a huge number of instances. Also an introduction of a single instance to the database can generate (by applying the present rules) a huge amount of logical consequents. Maintaining and enforcing integrity constraints, judging by the procedural equivalent of the declarative representation just given, over a reasonably populated database is a significant computational task. This looks, from a shallow perspective, like an operational problem. In fact if a predictable multi-dimensional data index is available it goes a long way to make integrity-constraint enforcement computationally acceptable.

From a more theoretical aspect we can reduce the scope of integrity constraint enforcement to checking only a minimum number of instances (i.e. those instances that are affected by the changes to the database state and the present integrity constraints). This technique is referred to as simplification and is motivated by the fact that if a database is consistent with respect to a set of integrity constraints then given an update to the database a set of simplified conditions (derived from the constraints and the update) have to be satisfied. The success of this technique lies in proving that the transformation of the constraints into simplified rules is safe and is complete for a given data-model building rules. There are three principle techniques cited in the literature: Nicolas *et al.* [NICOL78] and Lloyd *et al.*

[LLOYD85] for the theorem view, Kowalski *et al.* [KOWAL87] for the consistency view and Aquilino *et al.* [AQUIL97] adding new and restrictive rules to the deductive database.

7.2.2 Data Modelling Relationships

Relationships are an important element of any database. Like any other instances of the database, relationship instances are introduced, updated and purged – therefore a relationship instance has a life-span too. There are a number of relationship types each with a number of relationship properties, like for example a) arity, b) cardinalities and c) participation. The previous exposition of integrity constraints and relationships dealt with referential integrity. Some aspects of relationships are representable through combinations of integrity constraints; including referential constraints.

A basic representation of relationships and its set of instances follow. Let **RLSHP** be the name of a relationship, r_i one of the relations forming the relationship (the order of the names is to be used as a placeholder), and a_i one of the attributes that appertain to the relationship, also the ***dompk(*r_i*)*** reflects the primary key set of relation r_i:

$$\forall i_1 \ldots \forall i_m \forall x_1 \ldots \forall x_n \left(\begin{array}{l} \text{rlshp}(\text{i}_1, \ldots, \text{i}_\text{m}, \text{x}_1, \ldots, \text{x}_\text{n}) \\ \rightarrow \left(\begin{array}{l} i_1 \in \text{instance\_set}(r_1) \wedge \ldots \wedge i_m \in \text{instance\_set}(r_m) \wedge \\ x_1 \in dom(A_1) \wedge \ldots \wedge x_n \in dom(A_n) \end{array} \right) \end{array} \right).$$

$$\text{instance\_of\_rlshp} \in dompk(r_1) \times \ldots \times dompk(r_m) \times dom(A_1) \times \ldots \times dom(A_n).$$

$$\text{instance\_set\_of\_rlshp} \subseteq dompk(r_1) \times \ldots \times dompk(r_m) \times dom(A_1) \times \ldots \times dom(A_n).$$

7.2.2.1 Relationship Arity and Attributes

The arity of a relationship reflects the number of relations taking part and in the example it is m. Due to the relations and relationship method of representation chosen and the varied meaning of relationships we need to introduce a number of constraints for getting the exact meaning of a relationship under investigation. For example, a) primary-key constraint on each of the participating relations, b) primary-key constraint on the relationship relations, c) referential constraints between the relationship and each of the participating relations and d) other constraints on the relationship attributes – i.e. domain constraint on n attributes.

Each relationship can have any number of attributes that describes the relationship. Each relationship attribute has to have a domain associated with it and there might also be some integrity constraints.

In an n-ary relationship there are n cardinalities; one associated with each participating relation. For binary relationship the possible cardinalities are one-to-many (1-M), many-to-one (M-1), one-to-one (1-1) and many-to-many (M-N). An example of a binary 1-M follows, we have two relations namely department and student (i.e. **DEPT** and **STUD**), and a registration relationships (i.e. **REL_DEPT_STUD**) between their instances that requires each student instance to be related to one department instance. The structures and integrity constraints for specifying the example relationship follows:

Remark: dept and stud are two relation structures and rel_stud_dept_reg a relationship structure

$\text{dept}(d, dn)$.

$\text{stud}(s, sn)$.

$\text{rel\_stud\_dept\_reg}(s, d, rn)$.

Remark: primary key set (attribute d) constriant for department

$\forall d \forall dn \forall d' \forall dn' \left(\text{dept}(d, dn) \wedge \text{dept}(d', dn') \wedge d = d' \rightarrow dn = dn' \right)$.

Remark: primary key set (attribute s) constraint for student

$\forall s \forall sn \forall s' \forall s' \left(\text{stud}(s, sn) \wedge \text{stud}(s', sn') \wedge s = s' \rightarrow sn = sn' \right)$.

Remark: primary key constraint set for student & department relationship

$$\forall s \forall d \forall rn \forall s' \forall d' \forall rn' \begin{pmatrix} \text{rel\_stud\_dept\_reg}(s, d, rn) \wedge \text{rel\_stud\_dept\_reg}(s', d', rn') \wedge s = s' \\ \rightarrow d = d' \wedge rn = rn' \end{pmatrix}.$$

Remark: referential constraints between the relationship instances and student and department instances

$$\forall s \forall d \forall rn \begin{pmatrix} \text{rel\_stud\_dept\_reg}(s, d, rn) \\ \rightarrow \exists dn \left(\text{dept}(d, dn) \right) \wedge \exists sn \left(\text{stud}(s, sn) \right) \end{pmatrix}.$$

Remark: the actual relationship constraint that relates student to department instances follows

a student is registered in one department but

many students are allowed to register with a department

$$\forall s \forall d \forall rn \forall s' \forall d' \forall rn' \begin{pmatrix} \text{rel\_stud\_dept\_reg}(s, d, rn) \wedge \text{rel\_stud\_dept\_reg}(s', d', rn') \wedge s = s' \\ \rightarrow d = d' \end{pmatrix}$$

An example of a binary 1-1 relationship follows, we have the same two relations and a student representative relationship between their instances where this requires that each department is represented by single student and one student can represent one department only. The structures and integrity constraints for specifying this example are basically identical to the previous one except for the need to add other constraints to limit the oneness of the relationship from the department's side:

Remark: primary key constraint set (arbitary chosen attributes) for student & department relationship

$$\forall s \forall d \forall rn \forall s' \forall d' \forall rn' \left(\begin{array}{l} \text{rel\_stud\_dept\_rep}(s,d,rn) \wedge \text{rel\_stud\_dept\_rep}(s',d',rn') \wedge s = s' \\ \rightarrow d = d' \wedge rn = rn' \end{array} \right).$$

Remark: referential constraints between the relationship instances and student and department instances

$$\forall s \forall d \forall rn \left(\begin{array}{l} \text{rel\_stud\_dept\_rep}(s,d,rn) \\ \rightarrow \exists dn(\text{dept}(d,dn)) \wedge \exists sn(\text{stud}(s,sn)) \end{array} \right)$$

Remark: the actual relationship constraint that relates student to department instances follows

a student represents one department and

a department is represented by one student

$$\forall s \forall d \forall rn \forall s' \forall d' \forall rn' \left(\begin{array}{l} \text{rel\_stud\_dept\_rep}(s,d,rn) \wedge \text{rel\_stud\_dept\_rep}(s',d',rn') \wedge s = s' \\ \rightarrow d = d' \end{array} \right).$$

$$\forall s \forall d \forall rn \forall s' \forall d' \forall rn' \left(\begin{array}{l} \text{rel\_stud\_dept\_reg}(s,d,rn) \wedge \text{rel\_stud\_dept\_reg}(s',d',rn') \wedge d = d' \\ \rightarrow s = s' \end{array} \right).$$

As an example of a M-N relationship between relations student and department we can introduce the relationship of any student being able to register to any department's library. If we re-use the structure and integrity constraints found in the previous 1-N relationship we just need to do the following: a) introduce a relation for the relationship (e.g. **REL_STUD_DEPT_LIB**) and its relative primary and referential constraints; b) remove the last constraint that enforces the 1-N relationship.

To represent a ternary relationship requires more work. An example of a ternary relationship is: "Each student taking a unit sits at only one Lecture Theater and weekly time slot but can be at a different theater / slot for different units. For a given theater / slot, a student sits for only one unit. At a particular theater / slot there can be many students taking the same unit." Basically we have three relations (namely **STUDENT**, **LECTUREROOM** and **UNIT**) and one relation to enforce this N-1-1 relationship (i.e. **REL_S_LR_U_ASG**). We need the following constraints: a) primary key for each relation and a candidate key for the relationship; b) referential constraints from the relationship to each participating relations; and c) integrity constraints' that implement the meaning of the relationship.

Remark: unit, stud and lect_slot are three relation structures and

Remark: rel_u_s_lr_asg a relationship structure

$\text{unit}(u, un)$.

$\text{stud}(s, sn)$.

lect_slot(l, lsn).

$\text{rel\_u\_s\_lr\_asg}(u, s, l)$.

Remark: primary key set constraint for unit, stud and lect_slot (attribute u, s, l respectively)

$\forall u \forall un \forall u' \forall un' \left(\text{unit}(u, un) \wedge \text{unit}(u', un') \wedge u = u' \rightarrow un = un' \right)$.

$\forall s \forall sn \forall s' \forall sn' \left(\text{stud}(s, sn) \wedge \text{stud}(s', sn') \wedge s = s' \rightarrow sn = sn' \right)$.

$\forall l \forall ln \forall l' \forall ln' \left(\text{lect\_slot}(l, ln) \wedge \text{lect\_slot}(l', ln') \wedge l = l' \rightarrow ln = ln' \right)$.

Remark: referential constraints between relationship instances and unit, student and lect_slot instances

$$\forall u \forall s \forall l \left(\begin{array}{l} \text{rel\_u\_s\_lr\_asg}(u, s, l) \\ \rightarrow \exists un \left(\text{unit}(u, un) \right) \wedge \exists sn \left(\text{stud}(s, sn) \right) \wedge \exists lsn \left(\text{lect\_slot}(l, lsn) \right) \end{array} \right).$$

Remark: the actual ternary relationship constraint is expressed through two candidate key constraint

$$\forall u \forall s \forall l \forall u' \forall s' \forall l' \left(\begin{array}{l} \text{rel\_u\_s\_lr\_asg}(u, s, l) \wedge \text{rel\_stud\_dept\_reg}(u', s', l') \wedge u = u' \wedge s = s' \\ \rightarrow l = l' \end{array} \right).$$

$$\forall u \forall s \forall l \forall u' \forall s' \forall l' \left(\begin{array}{l} \text{rel\_u\_s\_lr\_asg}(u, s, l) \wedge \text{rel\_stud\_dept\_reg}(u', s', l') \wedge s = s' \wedge l = l' \\ \rightarrow u = u' \end{array} \right).$$

Relationships have another important property and this deals with specifying whether each instance of the relationship participating relation is to take part in a relationship instance or not–*total* versus *partial participation*. For a 1-N relationship there are four combinations: a) 1(total)-N(total); b) 1(total)-N(partial); c) 1(partial)-N(total); and d) 1(partial)-N(partial).

In the previous example between the relations student and department for the registration relationship it is reasonable to envisage that all students have to be registered but not all departments need to have students registered with them. To add the total condition to a relationship we need to introduce the following constraint to the relation that is in total participation: all instances of the student relation have to be registered with at least a department. The constraint in predicate calculus would be:

$$\forall s \forall sn \left(\text{stud}(s, sn) \rightarrow \exists d \exists rn \, \text{rel\_stud\_dept\_reg}(s, d, rn) \right).$$

Without any loss of generality this template is applicable to all types of binary relationships. A similar template is also available for ternary, and indeed n-ary, relationships.

7.3 Converting Integrity Constraints into Queries of Denials

To evaluate the static non-aggregated constraints as a database query we need to convert the integrity constraint representation (a closed first order predicate calculus) into a range-restricted denial in the form of (note that each $\boldsymbol{L_i}$ is a literal and variables are assumed to be universally quantified over the whole expression):

$$\leftarrow L_1 \wedge ... \wedge L_n.$$

Examples of the denials for the integrity constraints previously listed follow (these are based on transformation rules given in [LLOYD84]). In the case of the check constraint the integrity rule is converted into a query of denial as so:

$$\leftarrow student(x,y,z) \wedge y \neq 0 \; ... \; y \neq 5.$$

The use of denials is important, for example, during a design process. Assume we have the student relation and some instances that satisfy its basic specifications and we would like to introduce the check constraint. The evaluation of the query (of denial) would determine if the current state of the relation is already consistent with the specification of the check constraint or otherwise.

For the primary key set constraint we have:

$$\leftarrow student(x,y,z) \wedge sudent(x',y',z') \wedge x = x' \wedge (y \neq y' \vee z \neq z').$$

Remark: which is equivalent to the two queries

$$\leftarrow student(x,y,z) \wedge sudent(x',y',z') \wedge x = x' \wedge y \neq y'.$$

$$\leftarrow student(x,y,z) \wedge sudent(x',y',z') \wedge x = x' \wedge z \neq z'.$$

As for the functional dependence conversion into a denial we have:

$$\leftarrow \text{student}(x,y,z) \wedge \text{student}(x',y',z') \wedge x = x' \wedge z = z' \wedge y \neq y'.$$

In the case of constraint specification in which a literal contains free variables then another rule has to introduced to range restrict the denial. For example one of the constraints to specify a relationship was:

$$\forall s \forall sn (\text{stud}(s,sn) \rightarrow \exists d \exists rn \, \text{rel\_stud\_dept\_reg}(s,d,rn)).$$

The denial of this is:

$\leftarrow \text{stud}(s, sn) \wedge \neg \exists d \exists rn \, \text{rel\_stud\_dept\_reg}(s, d, rn).$

Remark: note variable s is free in the second literal.

Remark: we need to introduce a rule for this literal, namely

$\text{rr\_rel\_stud\_dept\_reg}(\text{s}) \rightarrow \exists d \exists rn \, \text{rel\_stud\_dept\_reg}(s, d, rn).$

Remark: rewrite the denial as a range restricted query

$\leftarrow \text{stud}(s, sn) \wedge \neg \text{rr\_rel\_stud\_dept\_reg}(s).$

7.4 Other Issues

There are a number of issues regarding design and implementation of integrity constraints. These include collective consistency of a set of constraints and redundancy of a subset of constraints. Furthermore integrity constraints, a topic of interest in this research can provide added knowledge to a query optimization process. Also of interest in this research is the extraction and encoding of constraints from design models, e.g. EERM.

7.4.1 Where to "attach" ICs

Given the way we have specified integrity constraints (i.e. to relations and to relationships without mentioning objects) it is questionable that although we have specified what we require we still need to convince ourselves that it is an acceptable representation. Specifically, if we want to up-hold the object homogeneity principle (mentioned in earlier chapters three and five) and modular design methodology then having integrity-constraint specifications spread over many artifacts does not augur well. Therefore if object encapsulation is to be respected then within an object's definition we need to include all of its constraints (static and aggregate, transitional and relationship constraints). This methodology does stretch other important aspects: a) redundancy is on the increase, for example in a binary relationships between two objects requires us to maintain two aspects of this relationship at each end; b) although our representation requires the introduction of another relation for each relationship it is not an absolute fact that the avoidance of this third relation is computationally cheaper; c) we are maintaining relations as first class citizens (i.e. objects) but reducing relationships (and their properties) as part of an object's composition.

7.4.2 Intra-IC Consistency

The way we have presented integrity constraints (i.e. a database state must be consistent to a set of schema's constraints) hide another facet of database consistency. Basically, are the

current sets of constraints consistent within themselves? In fact it is reasonable to expect that there exists a database state that satisfies the set of constraints. If no such database state is found or two constraints contradict each other then we have a set of unsatisfiable constraints.

Given that we have a logical representation of integrity constraints (through closed first order rules) then the resolution method is applied to the set of integrity constraints. If the empty clause is resolved then that means the set of constraints is inconsistent. There are two basic approaches to formalise this satisfiability (one due to Lloyd while the other due to Sadri and Kowalski is more general [SADRI87]). Let ***D*** be a database with a consistent completion ***comp(D)***. Constraint ***W*** is satisfied by ***D*** if ***W*** is a logical consequence of ***comp(D)***. ***W*** violates ***D*** if ***W*** is not a logical consequence of ***comp(D)***. If ***IC*** is a set of constraints (i.e. ***W***) then ***IC*** is satisfied by ***D*** if each constraint in ***IC*** satisfies ***D***. Note the theory framework is augmented with truth equality predicates.

In the following example shows two check integrity constraints in contradiction.

Remark: our example database has a single relation student with two attributes name (n) and sex (s)
Remark: $\text{stud}(n,s)$ – for this example there are no instances!
Remark: there are two integrity constraints on stud relation structure
$\forall n \forall s\left(\text{stud}\left(n,s\right) \rightarrow s = \text{male}\right).$
$\forall n \forall s\left(\text{stud}\left(n,s\right) \rightarrow s \neq \text{male}\right).$
Remark: the denials of the integrity constraints are
$\leftarrow \text{stud}\left(n,s\right) \wedge s \neq \text{male}.$
$\leftarrow \text{stud}\left(n,s\right) \wedge s = \text{male}.$
Remark: the completion of the database is
$\forall n \forall s\left(\neg \text{stud}\left(n,s\right)\right).$
Remark: apply the resolution (SLDNF) to the above expressions ($\text{comp(D)} \cup \text{I}$)

$$\frac{\text{stud}\left(n,s\right) \wedge s \neq \text{male} \quad \text{stud}\left(n,s\right) \wedge s = \text{male}}{s \neq \text{male} \wedge \text{stud}\left(n,s\right) \wedge s = \text{male}} \quad \text{unify on stud}\left(n,s\right)$$

□

7.4.3 Redundancy of ICs

A set of consistent integrity constraints may still not be optimal (or a minimal set). For example assume we have the student relation and two check constraints, e.g. **`STUD(X,Y,Z), Z=A`** and **`STUD(X,Y,Z), (Z=A OR Z=B))`**, then the latter subsumes the former. Basically we would want to avoid having a number of constraints derivable from others. This has a profound effect on the database operational and computational

requirements rather than on the logical consistency of the database state. The general problem is to reduce the original set of constraints into a logically equivalent set. The new set has to have the following properties: a) the original constraints are derivable from the new set; and b) evaluating the new set over the current database state is cheaper. Although parts of the general problem are easy to solve the problem itself, we conjecture, is a very hard computational problem. Furthermore, note that in general a reduction of constraints is applicable to a database state and not to all database states.

7.4.4 When to Apply IC Enforcement

It is expected that the state of a database is subject to frequent changes. This has a number of effects on enforcing the integrity constraints entrenched in the schema. Two effects of special interest are a) the mode and timing of checking a constraint during changes and b) the other is how computationally efficient can enforcement be.

The first effect deals with whether the integrity constraint is to be checked at the end of a database transaction or throughout the processing of the transaction. For example if we have a transaction that involves the introduction of a new department instance, a number of new student instances, and some registration relationship instances (some of which involve the new department) then if the referential constraint is to be checked on each instance introduction then it is apparent that the department's instance introduction has to be taken care of before the related students are introduced. Rather than try to resolve a transaction's dependencies it is best to reduce the problem of when to check for constraint violation. For example in this transaction it would be best to check it once at the end of the whole transaction. This is called the *deferred* approach. Likewise it is sometimes required that during a transaction some constraints are never broken. The check constraint on credit limit not being breached is a valid example of an *immediate* constraint violation requirement.

7.4.5 ICs in Query Processing

We have already presented the framework in which integrity constraint specification is an integral part of a database schema. A useful input into a schema includes a) physical storage requirements planning; and b) query processing and optimisation. For example, primary key and referential constraints point at the need for index structures that facilitate

introduction and maintenance of these constraints. As for query processing we want to use the knowledge about the schema, through structural dependencies, to reduce a query into a simpler (in terms of computational requirements) version of the original.

Let us consider a simple database with one relation, which has some structural constraints and a query over the same.

Remark : Students (attribute s) from department (d) is enrolled in a unit (u)
$\text{s\_d\_tu}(s,d,u)$.
Remark : the primary key set is made of attributes s and u
$\forall s \forall d \forall u \forall s' \forall d' \forall u' (\text{s\_d\_tu}(s,d,u) \wedge \text{s\_d\_tu}(s',d',u') \wedge s = s' \wedge u = u' \rightarrow d = d')$.
Remark : there is a functional dependency between s that determines d
$\forall s \forall d \forall u \forall s' \forall d' \forall u' (\text{s\_d\_tu}(s,d,u) \wedge \text{s\_d\_tu}(s',d',u') \wedge s = s' \rightarrow d = d')$.
Remark : we are interested to compute the following query
$\leftarrow \text{s\_d\_tu}(s,d,u) \wedge \text{s\_d\_tu}(s',d',u') \wedge s = s'$.

As the query is currently written a simplistic implementation is to run a double nested loop over the same relation; in relational database this is a join between two tables and specifically a 'self join'. If we consider the present functional dependency then we really do not need a join as the above query reduces into (e.g. a full relation scan):

$\leftarrow \text{s\_d\_tu}(s,d,u)$.

This optimisation technique is based on representing the query in tableau form and applying the *chase technique over the tableau* (the above query's tableau is shown next -see [AHOAV79] and [MAIER79]).

Step 1	Step 2	Step3
$\underline{\text{T}(S,D,U)}$	$\underline{\text{T}(S,D,U)}$	$\underline{\text{T}(S,D,U)}$
$\underline{\underline{b_1, b_4}}$	$\underline{\underline{a_2\ b_4}}$	$\underline{\underline{a_2\ b_4}}$
$a_1\ b_1\ b_2$	$a_1\ a_2\ b_2$	$a_1\ b_2\ b_4$
$a_1\ b_3\ b_4$	$a_1\ a_2\ b_4$	

In the first step we convert the original query into the standard form: i.e. the *distinguished variables* (those that are used in joins) are denoted by the $\boldsymbol{a_i}$ variables while the *non-distinguished variables* (or constants) are represented by $\boldsymbol{b_i}$ variables. For each introduced literal a row is created and absorbs the third literal as a join condition between the first two. The variable with the double underscore represents the query's output variables. Note that there should be at most one distinguished variable in each column. In the second step we apply the functional dependency (i.e. ***s*** determines ***d***) by the chase rule since the two tableau lines have the same distinguished variables in column ***s*** and therefore makes

$b_1 = b_3$ and therefore $b_1 = b_3 = a_2$. Since the chase rule has yielded two rows that are identical in terms of their distinguished variables then the third step reduces these "duplicate" rows into a single line. The third matrix is indeed an equivalent query without any joins.

The chase over the table given a set of dependencies, we have just seen, reduces the number of rows (i.e. joins) of a query. Is there a general procedure that guarantees this? The answer is negative and in fact it is a NP-complete problem. But if we restrict the dependencies to the functional ones only then the chase will yield a tableau, which is at most as expensive as the original query. It is important to emphasise that this technique applies to all of the query expression and therefore in contrast with the localised framework of other query optimisation techniques (most of these results are given in and attributed to [MAIER83]).

7.4.6 Other Issues - IC in CASE Tools / Database Design

One would expect to find some of these useful modelling and representation features available in prototypes and even more in commercial DBMS. This is not entirely the case. A case in point is integrity constraints where although the data definition language includes the constructs to specify constraints there is little support (or nothing) for the database administrator or database designer to reason and optimise the set of integrity constraints attached to a repository. Such an omission has effects not only on the logical validity of the design but also on the performance of the activities over the database.

7.5 Summary

Integrity constraints are an important element in database design. Furthermore many of these, especially the static ones, are easy to encode into first-order predicates and include them in denial queries. But their usefulness is not only limited to address data quality, as important as it is, but also aid in query optimisation for example.

In our study the extraction of constraints from design diagrams, inference of constraints from certain design patterns (e.g. *n*-ary relationships can have an array of functional dependencies), encoding in F-logic, are issues we address in the forthcoming chapters.

Chapter 8

Framework for an Object Database

8 –Framework for an Object Database Design

In the previous chapters we have synthesised an object-oriented data model, introduced the Datalog and logical programming, and gave details on database integrity constraints through their specification in first order logic.

In this chapter we give a specification of and implement a framework for an object database that supports the above synthesised data model. In the framework we find both an object base and support structures that maintain the object base. Furthermore static constraints are specifiable and enforceable by the framework. Toward this end F-logic is used for the design of a framework. F-logic allows for describing the structural schema including integrity constraints, the implementation of object's behaviour, and specifies the object base (i.e. the data of the database). Flora-2, a faithful interpretation of F-logic, has strong object-oriented features and logic programming that is extensively used to build the framework.

There are three aims for this platform. The first aim is for the object-oriented data model implementation. This entails an object with identity and its values in NF2 with logical identifiers, classes and object relationships through instance-of, classes and *ISA* relationships, data-type signatures, methods, and message passing. The second aim is for the framework to read an EERM diagram, validate it, and generate a schema in ODMG ODL's. (This is the scope of the following chapter). The third aim is to have a basis for more involved features that an object base requires: e.g. data-type checking and inference and query modelling.

The chapter uses data-modelling constructs described in chapters 2 through 5 as a foundation to build an object database. This foundation is also used as a basis for the all of the forthcoming chapters on data and query modelling work.

8.1 Basic Object Database Framework

The framework data requirements are best described top down. In it there are a few objects that are predefined and to which EERM design artefacts are made instances of; i.e. attached. Consequently the framework object base is partitioned through them. These objects' identifiers are: **`SCHEMA`**, **`CLASS`**, **`STRUCTURE`**, **`BD`**, **`DOMAIN`**, and **`CONSTRAINT`**.

The **SCHEMA** object is composed of a property to hold a schema name and a number of procedures, expressed in declarative Flora-2 [YANGG08]. These procedures take care of mapping schemas for example.

```
schema[ schemaname -> 'AfterScott' ].
```

An important object is **CLASS** whose instances are the design's classes and implemented with F-logic instance-of relationship (i.e. ":"). In turn if an application's domain class has instances of its own then these are related with an instance-of relationship too.

```
course      :class.       Remark course & dept are classes
dept        :class.
math        :dept.        Remark math & cs are instances of class dept
cs          :dept.
```

Furthermore if the application domain's classes have any *ISA* relationship between them then F-logic *ISA* relationship is used (i.e. "::"). As F-logic *ISA* allows for multiple inheritance and if this is not practised in the application domain design then a further constraint on the *ISA* between instances of **CLASS** has to be introduced. It has to be noted that F-logic semantics implement the directed graph nature of the *ISA* relationship.

```
student    ::person.      Remark student ISA person
lecturer   ::person.
ptstudent  ::student.
```

Another object is the **STRUCTURE** and it is used for building a complex value of an object. As seen in an earlier chapter description of F-logic an object composition is a set of properties and if a nested structure is required then a property needs to return another object. In this framework the later object is called a **STRUCTURE**. A **STRUCTURE** does have properties and an identity but it is part of the "holding" object. This nesting is not limited and therefore one can stimulate the NF2 with identifier structures explained earlier in Chapter 3 (section 3.3.2). These structures are also useful to implement weak instances as the weak instances actually depend on the controlling object. The following says that **JOB** is an instance-of **STRUCTURE** and two of its properties are given their signature.

```
job         :structure.
job         [jname{1:1}   *=> string,
             jbudget{1:1} *=> float].
```

The **BD & DOMAIN** objects basically partitions domains that are entertained in a design. In this framework the instances of **BD** are **INTEGER**, **FLOAT**, **STRING**, and **DATE_TIME**; these too are the basic domains mentioned in previous chapters. Enumerated domains are also entertained and are instances-of **DOMAIN** but have an added property called **ENUM** that hold its

elements as a set of strings. In the following example **SEX** is an instance-of **DOMAIN** while the second statement specifies what values are in its enumeration.

```
sex         :domain.
sex         [enum->{'male','female'}].
```

At this point we need to mention properties (or attributes and methods) and their attachment to instances of **CLASS** through signatures. The following states that object reference **DEPT**, an instance-of **CLASS**, has four inheritable attributes namely **DNAME**, **STAFF**, **SPONSORS** and **MAINOFFICE**. Their return data type is **STRING**, **LECTURER**, **COURSE** and **ADDRESS** respectively. The first is a basic domain, the next two are classes and the last a structure (i.e. **ADDRESS**). The braces directive, a slight deviation from F-logic syntax, explains the attribute's cardinality; e.g. **"1:1"** is one and only one, while "0:*" is any count from zero and upwards which corresponds to a set's cardinality.

```
dept      [dname{1:1}      *=> string,
           staff{0:*}      *=> lecturer,
           sponsors{0:*}   *=> course,
           mainoffice{0:1} *=> address].
```

Another object is called **INTEGRITY** and it partitions integrity constraints into a number of generic types: the most obvious are primary key (**PKCONSTRAINT**), functional dependence (**FDCONSTRAINT**), and referential integrity (**RFCONSTRAINT**). All of these are really constraints from the application domain and every generic constraint is expected to have a number of instances that are required by the application domain. Any other constraints of the framework, for example enforcing a single inheritance regime between instances of **CLASS**, should be implemented as rules instantiating the object **INTEGRITY**.

```
pkconstraint    ::constraint.

pkconstraint    [classname{1:1} *=> class,
                 attrlist{1:*}  *=> string].

pk2:pkconstraint[consname  -> 'dept pk',
                 classname -> dept,
                 attrlist  -> {dname}].
```

A useful addition to the framework, recommended by Kifer *et al.* [KIFER95], is to have a rule that returns on object's identity as a value. This syntactic sugar at times makes expressions more concise. Also it is useful in path expressions.

```
?X[self->?X].
```

In Flora-2 rules and queries any token starting with "**?**", for example "**?X**", is a variable. While token "**?-**" at the beginning of a line denotes the start of a query and the latter terminates

with a full stop, for example "**?- ?X:CLASS.**". The token "**REMARK**" denotes verbose comments and is delimited by end of line. The "Yes" denotes successful query evaluation.

A number of 'views' are available in the framework too. For example the following rule makes instances of **CLASS** have a property, called **DIRECTASC**, which contains a direct ancestor if present. The query then asks for **CLASSES** that have **DIRECTASC** defined and what are the pairs.

```
?D[directasc  -> ?C]   :-                          Remark Rule definition for
     ?D::?C, ?D:class, ?C:class,                   Remark  directasc
     If ( ?T:class, ?D::?T, ?T::?C )
     then (false)
     else (true).

?- ?C:class, ?C[directasc->?A].
?C = lecturer  ?A = person
?C = ptstudent ?A = student
?C = student   ?A = person
Yes.
```

Similar to **DIRECTASC** is **DIRECTDESC** which lists a **CLASS** instance direct descendants. This property is useful to define the instantiating class of an object; for example an instance-of **PTSTUDENT** is also an instance-of **STUDENT** and **PERSON** by inference, but an instance-of **STUDENT** is not a **PTSTUDENT**. To know the class which created an instance we have an **INSTCLASS** property which is define by the following rule. We reiterate that in F-logic: the query **A:C** implies that **A** is in the deep extent of **C**; and the query **C2::C1** implies that **C2** is a subclass of **C1** directly or by transitivity of the *ISA* relationship.

```
?I[instclass->?C]:-?I:?C,?C:class, not ?I:?C.directdesc.

?- poca:?C,?C:class.    Remark object id poca is an instance-of which class?
?C = person
?C = ptstudent
?C = student
Yes.

?- poca[instclass->?C]. Remark poca is an instance-of which instant. class?
?C = ptstudent
Yes.
```

8.1.1 Data Requirements of Framework

The framework objects and their instances are related in a number of ways. The following is a description and figure 8.1 contains an EERM that depicts these.

Each instance of a class and structure has many attributes while an attribute must be related to only one instance of a class or a structure. Also some instances of a class are related by a specialisation relationship; but a class instance can only have at most one ancestor. Each attribute has a signature that includes name, cardinalities and return type details. The

return types can be any one of the following: a class, a structure, a basic domain, and an enumerated domain.

A constraint is specialised into primary key (marked PK in figure 8.1), functional dependence (marked FD), referential constraint (binary and no attributes – marked RC ODMG), and referential constraint (other than binary with no attributes – marked RC). This EERM *ISA* relation is disjoint and total.

The primary key constraint instances are related to attributes through a one-to-many (but all attributes must be from the same class or structure).

Either referential constraint is composed of exclusive elements that represent each degree. For example a binary relationship has two element components whereas a ternary has three.

If the referential constraint is binary and without attributes (i.e. RC ODMG) then each element is related to a class attribute (i.e. the foreign key) and a relationship to the class it is pointing to. Each element has properties for its cardinalities range, lower and upper limits, and an arbitrary sequential number.

In the case of a binary referential constraint (i.e. RC) element (other than binary with no attributes) they are related by structures through a many-to-one relationship and also have two additional details. The first is the attribute in the structure that is a foreign key and the second is to which class or structure it is pointing to. Also each element has its cardinalities range, lower and upper limits, and an arbitrary sequential number.

In the case of an *n*-ary referential constraint with attributes the entity (i.e. RC) is related with a weak relationship to many attribute signatures.

The functional dependence (i.e. FD) between attributes is implemented with a many-to-many relationship adorned with an attribute that indicates whether each attribute is on the left or right of a functional dependence.

8.1.2 Logic Programming

Logic programming, based on the resolution inference, is available in Flora-2. The following example highlights its use to develop our object-database framework. As an example, an implementation of an integrity check denial is described; the constraint deals with our

insistence that any instance of a class has to be an instance of at most one class. The main part of the denial checks for an object instance (`?I`) being related (i.e. by ":") to two objects. The latter objects are distinct class instances and are not related through the *ISA* relationship (i.e. ": :"). If such instances exist then the method collects and reports them and finally fails.

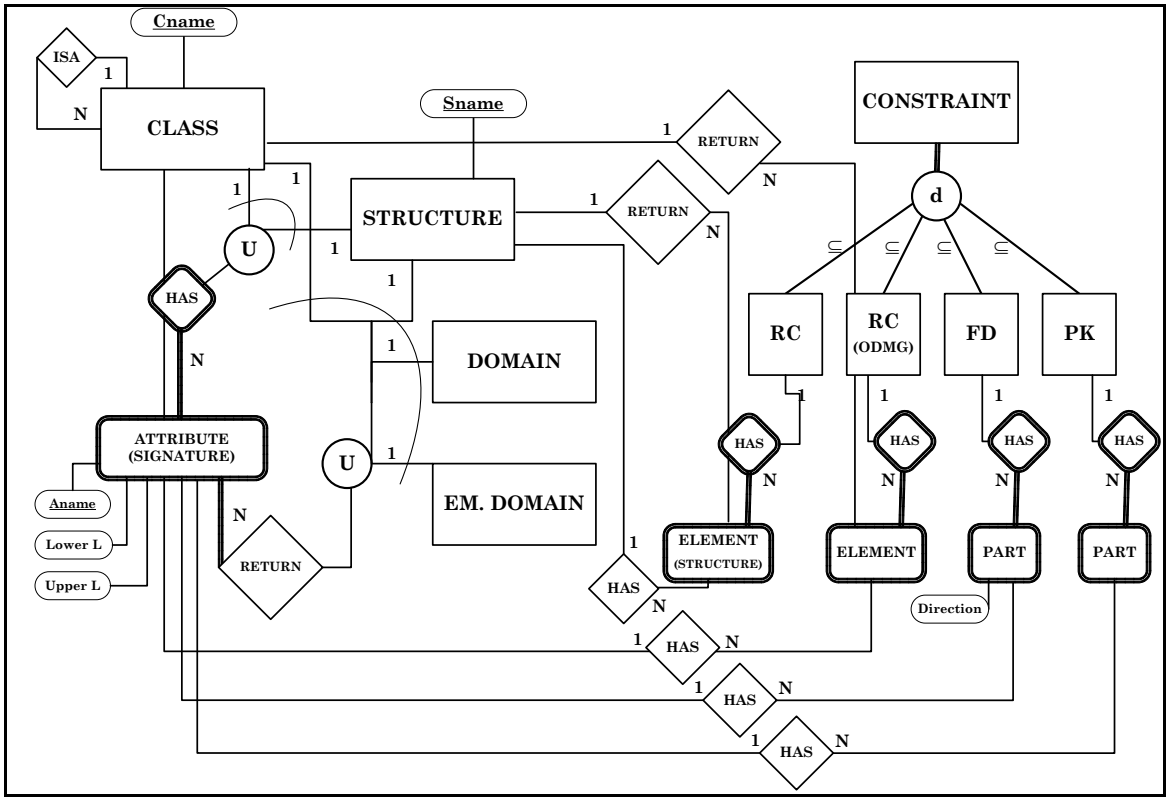


Figure 8.1 – High-level Data relationships for Object Database Framework

```
constraint[%ic_schema_no_mio]:-
    write('-- ic schema no multiple instance-of ')@_prolog,
    writeln('relationships for instances of classes:')@_prolog,
    ?C1:class, ?C2:class, ?C1!=?C2, not(?C1::?C2), not(?C2::?C1),
    if (?I:?C1, ?I:?C2)
    then (?L=collectset{?I | ?C1:class,?C2:class, ?C1!=?C2,
                             not(?C1::?C2), not(?C2::?C1), ?I:?C1, ?I:?C2 },
          writeln('IC Broken! Problem with: ')@_prolog,
          writeln(?L)@_prolog,
          false)
    else (writeln('OK!')@_prolog).
```

The following session script shows how one can use this method to sift out such instances.

```
andy:ptstudent.                          Remark object andy is an instance of
andy:unit.                               Remark  ptstudent and unit
javaprog:project.
javaprog:unit.

?- constraint[%ic_schema_no_mio].
-- ic schema no multiple instance-of relationships for class instances:
IC Broken! Problem with:
[andy, javaprog]
No.
```

8.2 Schema in F-logic

An example schema is required to encode in the object database framework. The application domain deals with a university setup that caters for courses and projects. Colloquially we call this schema 'afterScott'. The main requirements follow and a graphical interpretation is in figure 8.2.

The main players are lecturers, students and part-time students. These are related with two disjoint and partial *ISA* relationships. Also each person has a set of communication lines but some communication lines can be unassigned. Each lecturer can have a number of degrees. A project is composed of a number of jobs and each job has a budget. Project instances have a number of relationships: a lecturer can lead a number of projects and each must have one leader; each project can have a number of persons working on it and conversely a person can work on many projects; some projects are a continuation of other projects.

The university is split into departments and each department has a number of lecturers but each lecturer works in one department. A department has an exclusive address. Each department can co-sponsor, with other departments, a number of courses for students to enrol on. Each student can enrol in one course at a time. It is expected that many students enrol on a course. Also a course has a number of units that students are required to take up and a unit is included in many courses. Each unit has one coordinator, a lecturer, but many lecturers can be involved in teaching it from year to year.

Every student's performance for a transcript is to be kept: that's a student registers for many units and a unit is taken by many students. For each registration the grade awarded is recorded.

8.2.1 Asserting EERM Constructs into Flora-2

Each EERM diagram is encoded into some syntax along the artefacts given in Fig. 8.1 above. Many EERM design tools offer a function to export a diagram into an ASCII file and an XML encodings. The encoded syntax has to be read into the framework and assigned into its structures (e.g. **`CLASS`**, **`STRUCTURE`**, **`CONSTRAINTS`**). In this case the encodings are hard coded into the framework rather than read from some exported file.

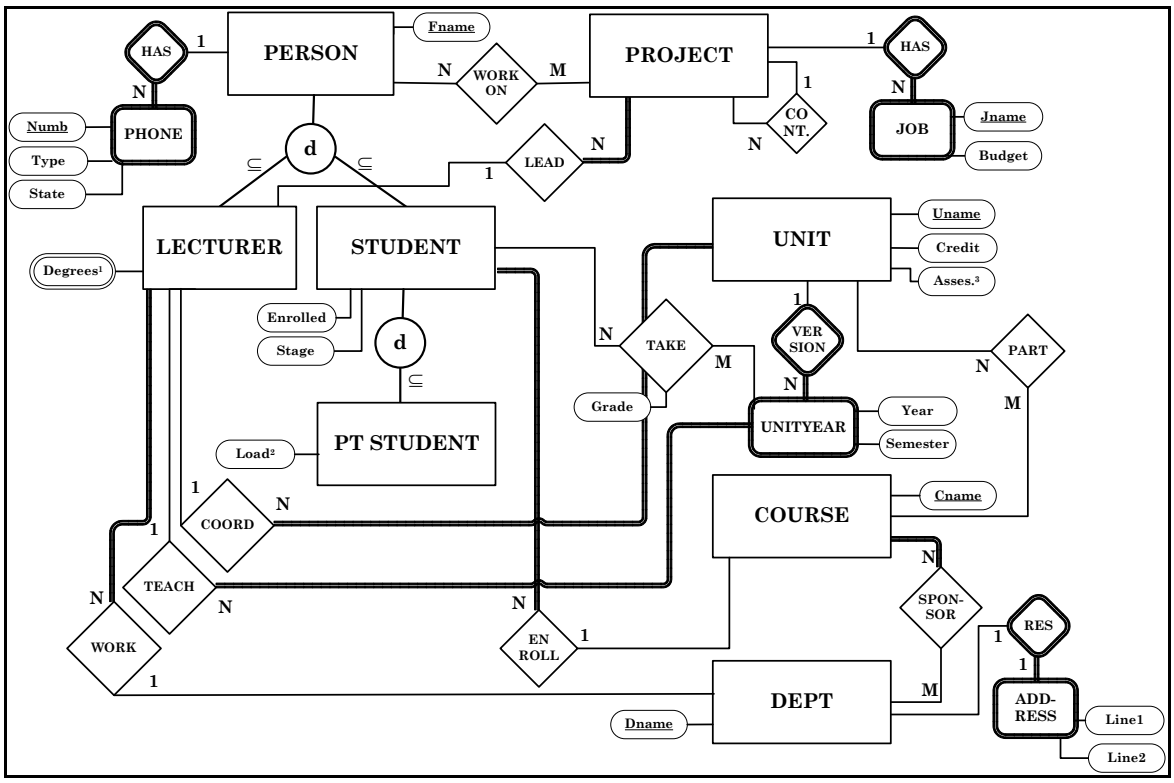


Figure 8.2 – High-level example object database design – 'AfterScott'

What follows is a sequence of EERM encodings into Flora-2 facts and assertions. We start with entities and attributes, *ISA* relationships, primary constraints, weak entities, and finally EERM relationships.

8.2.1.1 Entity template

Every diagram entity is asserted as an instance-of object **`CLASS`**. The name of each entity, which is unique in a diagram, is transcribed as its logical identifier. The template for each entity is:

```
entity_name : class.
```

8.2.1.2 Attribute's template

As we had presented earlier an entity's attributes can be classified by three selectors: simple *vs* structured, single *vs.* multiple, and stored *vs.* computed. We assume to require only stored attributes and computed ones are taken care of by query constructs.

A simple attribute, in an EERM diagram, would have a name and a data type (first template below). If the data type is not a basic domain but an enumerated type then the latter needs to be defined (second template). It is assumed that an enumerated type is a set of strings. All simple attributes are assigned as inheritable attributes (i.e. "`*=>`"). In this template it is being

assumed that attribute is a single value – i.e. takes one and only one value (denoted by `{1:1}` for lower and upper cardinality constraints).

```
entity_name [ attribute_name{1:1} *=> basic_domain ].
```

```
enumerated_domain_name : domain.
enumerated_domain_name [ enum -> {'element1', … , 'elementN'} ].

entity_name [ attribute_name{1:1} *=> enumerated_domain_name ].
```

If an attribute takes a structured nature then we need to introduce a structure definition for it and assign it to the attribute's return type. Recall that an ERM diagram only allows one level of nesting in a structure. The following template shows the artefacts required; note that domain is either a basic domain or an enumerated domain (in which case the enumerated domain template is used).

```
structure_name : structure.
structure_name [ attr_name1{1:1} *=> domain1, … ,
                 attr_nameN{1:1} *=> domainN ].

entity_name [ attribute_name{1:1} *=> structure_name ].
```

It was assumed above that attributes take a single value but it is known that the ERM attributes can take a set of values from a domain. To support this an attribute's cardinality is changed from "`{1:1}`" to "`{0:*}`" with the latter reflecting zero to many values assignable to an attribute. The following templates show the difference between single and multiple valued attributes. This construct is applicable to both simple and structured attributes.

```
entity_name [ attribute_name{1:1} *=> domain ].

entity_name [ attribute_name{0:*} *=> domain ].
```

8.2.1.3 ISA template

The *ISA* relationship is between entities but its varieties really constrain what relationships are taken up and what affect it has on the instance-of relationship. The basic *ISA* relationship template is a quite straightforward assertion in Flora-2.

```
entity_name1 :: entity_name2.
```

As it is our opinion that an EERM entity hierarchy is expressed in single inheritance only and that both EERM and F-logic allows multiple inheritance then instances of **`CLASS`** should be checked against taking up multiple inheritance. The multiple inheritance denial check is attached to the object **`CONSTRAINT`** and is embedded in a method called **`%IC_SCHEMA_NO_MI`**. On its invocation the *ISA* assertions are tested and if it is found to be the case that any **`CLASS`** instance inherits from more than one **`CLASS`** instance the situation is flagged (i.e. *failed*) and

any **CLASSES** with multiple inheritance are listed–this is taken care of by Flora-2 aggregate construct *collectset*. The pattern that catches multiple inheritance is simple: is there a **CLASS** (**?C**) that inherits from two non *ISA* related **CLASSES** (**?P1** and **?P2**)? Additional conjuncts are used to address distinctness: limit scope as **?C**, **?P1** and **?P2** need to be **CLASSES**; that symmetrical instantiations are attenuated; and the spurious case ignored (i.e. where **?P1** and **?P2** are identical). The procedure uses the syntactic sugar implementation of the 'if then else' rule found in Flora-2.

```
constraint[%ic_schema_no_mi] :-
    writeln('-- ic schema no multiple inheritance between classes:')@_p,
    ?C:class, ?P1:class, ?P2:class, ?C::?P1, ?C::?P2, ?P1!=?P2,
    if ( not(?P1::?P2), not(?P2::?P1) )
    then ( ?L=collectset{?C | ?C:class, ?P1:class,
                              ?P2:class, ?C::?P1, ?C::?P2,
                              ?P1!=?P2, \+(?P1::?P2), \+(?P2::?P1)},
             writeln('IC Broken! Problem with: ')@_prolog,
             writeln(?L)@_prolog,
             false)
    else (   writeln('OK!')@_prolog).
```

To test the procedure one needs to adjust the class *ISA* assertions with a multiple inheritance class; e.g. say **PTSTUDENT** inherits from **STUDENT** and **DEPT**.

```
student    :: person.
lecturer   :: person.
ptstudent  :: student.
ptstudent  :: dept.
```

The procedure invocation would output the offending classes with multiple inheritance and then fail.

```
?- constraint[%ic_schema_no_mi].
// ic schema no multiple inheritance between classes:
IC Broken! Problem with: [ptstudent]
No.
```

The EERM description of *ISA* includes three varieties: whether it is disjoint *vs* overlapping, total *vs* partial, and semantic *vs* predicated defined.

The overlapping *ISA*, without doubt a useful design construct, requires extensive DBMS support to implement. For example, in the relational data model, one implementation would need views with triggers that can update a view's base tables. ODMG is still inadequate in this area and consequently overlapping *ISAs* are to be rejected for the time being. To enforce disjointness between sibling classes an integrity constraint is implemented with a method called **%IC_CLASS_DISTINCT_ISA** found in object **CLASS**. The method works by checking if any two distinct and direct subclasses, say **?C1** and **?C2**, of a given class **?C**, share any

instance – i.e. object **?I**. This constraint is really a more specific example of **%IC_SCHEMA_NO_MIO** found in object **CONSTRAINT** and described earlier.

```
class[%ic_class_distinct_isa(?C)] :-
    ?C1:class, ?C2:class, ?C1!=?C2, ?C1.directasc=?C, ?C2.directasc=?C,
    if (?I:?C1,?I:?C2,!)
    then (  ?L=collectset{?I2 | ?C11:class, ?C22:class, ?C11!=?C22,
                        ?C11.directasc=?C, ?C22.directasc=?C,
                        ?I2:?C11,?I2:?C22 },
            writeln('IC Broken! Problem for class: ')@_prolog,
            writeln(?L)@_prolog,
            false)
    else (   writeln('OK!')@_prolog).
```

Another *ISA* constraint is the total. This implies that the superclass cannot have instances. The generic constraint for the denial of this pattern is implemented in method **%IC_CLASS_TOTAL_ISA** in object **CLASS**. The method checks if a given **CLASS** instance, method argument **?C**, has any subclasses (actually this shouldn't be necessary) and if there are instances that have **?C** as their instantiating class then the constraint is broken.

```
class[%ic_class_total_isa(?C)] :-
    if ( ?C:class, ?C1:class, ?C!=?C1, ?C1::?C, ?I:?C, ?I.instclass = ?C, !)
    then ( ?L=collectset{?I2 |  ?I2:?C, ?I2.instclass = ?C },
           writeln('IC Broken! ISA implies it must not create objs: ')@_p,
           writeln(?L)@_prolog,
           false )
    else ( writeln('OK!')@_prolog ).
```

For predicate defined *ISA* we need to add a broader template to the *ISA* assertions. First we need to identify the attribute on which the predicate is defined, and the respective expression (here we assume the predicate has the form – i.e. **ATTRIBUTE = CONSTANT**). Secondly we have to ensure that instances do not change this value – this is implemented with a check constraint. The following constructs show what the templates need to produce for having a class **PERSON** specialized into **MALE** and **FEMALE** with instances of subclasses determined by predicate **GENDER**. Of special significance here is the inheritance properties (i.e. "***->**") which implies that any instance-of **MALE**, for example, would inherit the property **GENDER -> 'MALE'**. The last two lines represent the denial queries for the check constraint.

```
person [ gender{1:1} *=> sex ].

male::person.                    female::person.
male   [ gender *-> 'male'].     female [ gender *-> 'female'].

?- ?I:male,?I[gender->?S],?S!='male'.
?- ?I:female,?I[gender->?S],?S!='female'.
```

The template, including *ISA* assertions presented earlier, follows. Note that object **CHECK** is a subclass of constraint and **CH_ENTITY11**, for example, is an instance-of **CHECK** object. The **CHECK** object signature follows:

```
check [classname{1:1} *=> class,
       attrlist{1:1}  *=> string,
       value{1:1}     *=> string].
```

The **CH_ENTITY11** constraint enforces the rule that **ENTITY11** has an attribute **ATTRNAME** whose value must be **'STRING11'**.

```
entity11 :: entity1.
entity12 :: entity1.

entity1  [ attrname{1:1}*=>string ].

entity11 [ attrname *-> 'string11' ].
entity12 [ attrname *-> 'string12' ].

ch_entity11 : check.
ch_entity11 [ classname->entity11, attr->attrname, value->'string11' ].

ch_entity12 : check.
ch_entity12 [ classname->entity12, attr->attrname, value->'string12' ].
```

For scrutinising the above check constraint a method, called **%CHECK_CONS**, of object **CHECK** is called. The method takes an argument that represents the actual check constraint to be checked. The check constraint implement here is trivial and is based on string equality. The idea is to take the details from a check instance, e.g. **CH_CHECK11**, and see if any object breaks this constraint. The following invocation checks if all **TELEPHONE** instances are of '**CELL'** .

```
check :: constraint.
constraint[consname{1:1}*=>string].
check [classname{1:1}*=>class, attrlist{1:1}*=>string, value{1:1}*=>string].

check[%check_cons(?Ch)] :-
    writeln('-- check cons  attr = value ')@_prolog,
    if   ( ?I:?Ch.classname,?I[?Ch.attr->?T],?T != ?Ch.value, ! )
    then ( ?L=collectset{?I2 | ?I2:?Ch.classname,?I2[?Ch.attr->?T2],
                               ?T2 != ?Ch.value },
           writeln('IC Broken! Problem with: ')@_prolog,
           writeln(?Ch)@_prolog,
           writeln(?L)@_prolog,
           false)
    else ( writeln('OK!')@_prolog).

ch1:check[consname-> 'is cell',
          classname-> telephone, attr->tptype, value->'cell'].

t789012:telephone  [ tpname->'t789012',  tptype->'cell'  ].
t901234:telephone  [ tpname->'t901234',  tptype->'xcell' ].

?- check[%check_cons(ch1)].
-- check cons  attr = value
IC Broken! Problem with:
ch1
[t901234,t99123456]
No.
```

Although the *ISA* assertion templates have been given, what is still required are details of the *ISA* varieties that are read from the EERM designer. An object has been created, called **ISAPROPERTY**, which is a subclass of **CONSTRAINT** and with a signature that follows. Single valued attribute **PARENTCLASS** and set based attribute **ISACLASS** holds the *ISA* relationship, the next three attributes denote the specific variety of the *ISA* relationship. The last two attributes are optional and used only to describe a predicate define *ISA* relationship.

```
isaproperty  [ parentclass{1:1}              *=> class,
               isaclass{1:*}                 *=> class,
               disj_over_flag{1:1}           *=> string,
               totl_part_flag{1:1}           *=> string,
               sema_pred_flag{1:1}           *=> string,
               predicate{0:1}                *=> string,
               pred_value_list(class){0:*}  *=> string ].
```

An instantiation of **ISAPROPERTY** class to implement a partial and semantic *ISA* relationship anchored at **PERSON** follows:

```
isaperson:isaproperty [ parentclass->person, isaclass->{lecturer, student},
                        disj_over_flag -> 'disjoint',
                        totl_part_flag -> 'partial',
                        sema_pred_flag -> 'semantic' ].
```

In case of a predicate defined *ISA* relationship, as exemplified with **PERSON**, **MALE** and **FEMALE** given previously, an instantiation **ISAPROPERTY** object would be:

```
isapersongender:isaproperty
     [ parentclass -> person,
       isaclass -> {male, female},
       disj_over_flag->'disjoint',
       totl_part_flag->'total',
       sema_pred_flag->'predicate',
       predicate -> gender,
       pred_value_list(male)->'m',
       pred_value_list(female)->'f'].
```

Another two restrictions are required to maintain a reasonable class hierarchy. Firstly, we can arbitrarily disallow instances of structures to participate in *ISA* relationships. Secondly, a class instance can only be a parent class of one *ISA* relationship.

8.2.1.4 Primary Key constraint template

Each entity has a primary key made up from a set of its attributes. Keys are a constraint too and therefore **PKCONSTRAINT** *ISA* **CONSTRAINT**. **PKCONSTRAINT** signature follows:

```
pkconstraint    [classname{1:1}*=>class, attrlist{1:*}*=>string].
```

A definition of a primary key has a name, a class to which it is applicable, and the set of attributes it is made from. These details are readily available in an ERM diagram. The following says that object **PK1** is an instance-of **PKCONSTRAINT** and it describes entity **COURSE** primary key set as being made up of a single attribute **CNAME**.

```
pk1:pkconstraint.
pk1 [ consname  -> 'course pk',
      classname -> course,
      attrlist  -> {cname} ].
```

To check that indeed for the deep extent of a class, e.g. **COURSE**, has no key duplication a general method called **%PK_CONS_ONE** is invoked and takes a primary key instance identifier as an argument. The method checks if any two distinct instances of the relevant deep extent, denoted by the path expression **?PK.CLASSNAME**, have their primary key attribute values, denoted by path **?IN.?PK.ATTRLIST** , equal.

```
pkconstraint[%pk_cons_one(?Pk)]:-
    writeln('-- pk cons (one) ')@_prolog,
    if   (?I1:?Pk.classname, ?I2:?Pk.classname, ?I1 != ?I2,
          ?A=?Pk.attrlist, ?I1.?A = ?I2.?A, ! )
    then (?L=collectset{?Key | ?I11:?Pk.classname, ?I21:?Pk.classname,
                               ?I11 != ?I21, ?A1=?Pk.attrlist,
                               ?Key=?I11.?A1 , ?I11.?A1 = ?I21.?A1  },
          writeln('IC Broken! Problem with: ')@_prolog,
          writeln(?Pk)@_prolog,
          writeln(?L)@_prolog,
          false)
    else (writeln('OK!')@_prolog).
```

A typical runtime session using **PK1** above would be:

```
?- pkconstraint[%pk_cons_one(pk1)].
-- pk cons (one)
OK!
Yes.
```

Say we pollute the sample data with incorrect data, for example two courses have the same name, and then this invocation would run and response on the following lines:

```
?- pkconstraint[%pk_cons_one(pk1)].
-- pk cons (one)
IC Broken! Problem with:
pk1
[BSc Engineering]
No.
```

Unfortunately **%PK_CONS_ONE** has a flaw; it provides for singleton attribute set which is not the general case. Another method, **%PK_CONS_MANY**, is available that takes care of this and follows the structure of a universal quantification query (i.e. three levels and double negation). Basically we are looking to show that for primary key violation it is not the case that any two instances do not have a different value in any attribute of the key set. The technique of reading and appending path expressions, adopted in the first version, is kept in this method too.

```
pkconstraint[%pk_cons_many(?Pk)]:-
    writeln('-- pk cons (many) ')@_prolog,
    if (not %pk_cons_many_mid(?Pk))
    then (writeln('Ok ')@_prolog)
    else (writeln('-- problem with (many) ')@_prolog).

    %pk_cons_many_mid(?Pk):-
        ?I1:?Pk.classname,
        ?I2:?Pk.classname,
        ?I1 != ?I2,
        not %pk_cons_many_in(?I1,?I2,?Pk).

        %pk_cons_many_in(?I1,?I2,?Pk):-
             ?A=?Pk.attrlist,
             ?I1.?A != ?I2.?A.
```

It has to be confirmed that a primary key definition is applicable to a class deep extent.

In this section we implicitly assumed that a key's attributes are value based; that is an attribute takes a value from a defined basic domain. But this need not be the case in this data model as an attribute can take an object reference (a logical identifier) that is an instance-of a class rather than a basic domain. In this situation the above routines still implement the primary key constraint even if any of the attributes take an identifier.

Candidate keys, although not depicted in the EERM diagram, are very useful in logical design. There specification and checking is identical to primary key other than their instantiating class which is **CKCONSTRAINT**.

8.2.1.5 Weak entities and weak relationships template

Weak entity instances have a dependent existence on a 'normal' entity instance through a weak relationship. A solution adopted here for this data pattern is to relate weak instances by an attribute those return data type is a tuple structure. In our framework we have a specific object, called **STRUCTURE**, whose structure instances are tuple definitions.

Our example schema has some weak entities such as **JOB, UNITYEAR** and **ADDRESS**, and **JOB** is weakly related to entity **PROJECT**. The following shows the ERM encoding required through instance-of and signatures assertions, and also some data examples. Since this weak relationship is 1(p)-N(t) (or 1(t)-N(t)), then on the side of **PROJECT** the attributes **JOBS** implements a zero to many cardinality; while on the side of **JOB** structure it takes one, and only one, reference to a **PROJECT** instance.

```
project : class.
job     : structure.

project  [ pname{1:1} *=> string, jobs{0:*}    *=> job   ].
job      [ jname{1:1} *=> string, jbudget{1:1} *=> float,
           jproj{1:1} *=> project ].

proj1    [ pname -> 'Project 1',  jobs    -> { j1, j2 } ].
j1:job   [ jname -> 'Job 1',      jbudget -> 123.4,  jproj -> proj1 ].
```

The reader can notice that there is a missing referential constraint to enforce the weak relationship present. The ERM designer tool needs to export this constraint too and the framework reads it into a referential constraint whose partial signature is provided below. The **RELTYPE** attribute is qualified with a 'weak' string. The attribute **RELSEQ**, which is a string, uniquely identifies EERM relationships. Since a weak relationship is binary then two **COMP** (i.e. components) attributes values are defined. As for the single side **COMP** attribute it takes six arguments and returns a **STRUCTURE** instance (where the weak entity structure is defined); whereas the many side takes the six arguments too but returns a class. The arguments are the name of the entity (or weak entity); the attribute that holds the relationship instance, role name, the lower and upper cardinality of the relationship, and finally a distinctive sequence number. The following is an ISA and referential constraint signature.

```
refconstraint::constraint.

refconstraint [ reltype{1:1} *=> string, relseq{1:1} *=> string,
                comp(class, string, string,
                    cardinality, cardinality, integer){0:1} *=> class,
                comp(class, string, string,
                    cardinality, cardinality, integer){0:1} *=> structure ].
```

To encode the weak relationship for **JOB** the following object in Flora-2 is required.

```
rc12:refconstraint.
rc12[ consname-> 'project jobs', relseq->'x1'].
rc12[ reltype-> 'weak', comp(project, jobs, 'hasJobs', 0, '*', 1) -> job,
                        comp(job, jproj, 'isPartOf', 1, 1, 2) -> project ].
```

The next section gives details as to how referential constraints in our framework are checked against the object collection.

To ensure that no weak entity instance lives on its own, a transitional constraint is indicated over a static constraint. In practice this is embedded in a database trigger associated with the entity that controls the weak entity. In the above example the class **PROJECT** would have a trigger that on deletion of a **PROJECT** instance a cascade of deletes are executed for purging the weak instances of **JOB** assigned in the **JOBS** attribute. ODMG does not directly support

triggers; a workable solution is for the designer to add incremental code (i.e. to delete any related **JOB** instances) to the **PROJECT**'s delete method inherited from ODMG's ancestral class.

There is another scenario that needs addressing; since in our framework instances of a **STRUCTURE** are objects, referential constraints are not sufficient to inhibit 'loose' instances. If a **STRUCTURE** is not a return type of a method, i.e. not listed in a weak relationship, then it can have instances without breaking any referential constraint. What is required is an integrity constraint that searches for **STRUCTURES** not mentioned in any relationship and then checks if it contains any instances. The method **%IC_STRUCTURE_NOLOOSEEND** of object **STRUCTURE** checks if any instance-of **STRUCTURE**'s that is not covered by any return data type of an inheritable attribute (i.e. in a signature of the form **?C[?A *=> ?S]**) and has instances. It is important to note the versatility of mixing signature, instance-of, and class based predicates in a single query.

```
structure[%ic_structure_nolooseend]:-
        writeln('-- ic weak entity instances cannot live on their own')@_p,
        if    ( ?S:structure, ?_IS:?S, not ?_C:class[?_A*=>?S], !)
        then ( ?L=collectset{?Ss | ?Ss:structure, ?_ISs:?Ss,
                                   not ?_Cs:class[?_As*=>?Ss]},
               writeln('IC Broken! Problem with: ')@_prolog,
               writeln(?L)@_prolog,
               false )
        else ( writeln('OK!')@_prolog).
```

(Note: variables starting with an underscore, e.g. "**?_C**" above, denote anonymous variables.)

The invocation of the method that follows derives that a structure called **LOOSENED** has instances and yet it is not being referred to by any inheritable attributes.

```
?- structure[%ic_structure_nolooseend].
-- ic no structure instance is not assigned to an attribute
IC Broken! Problem with:
[looseend]
No.
```

The last issue is the primary key definition of a weak entity. If the **STRUCTURE**'s instances have a primary key set that ensures uniqueness across all its extent then a normal primary key constraint definition, as presented in a previous section, is adequate. If it is not the case then one solution is to append the object identifier of latter to each weak instance. For example in weak entity **JOB** the attribute **JNAME**, although unique in any **PROJECT** instance, it is not unique across all instances of **JOB**. To rectify one appends **JPROJ** attribute to the key set – recall its signature enforces **{1:1}** cardinality.

```
pk8:pkconstraint [consname-> 'job w pk',
                  classname-> job, attrlist->{jname, jpoj]}].
```

8.2.1.6 Binary Relationship templates

Relationships are an important part of ERM diagrams. These have many varieties and therefore need a flexible approach to encode and generic methods to enforce. In this subsection we start with a binary relationship without any attributes and then consider the effects of adding attributes to it. In subsequent sections the more involved relationship designs are encoded. We also need to subsume the previously introduced template for a weak relationship.

The signature for a referential constraint has already been given (i.e. in the weak entity section). For a normal relationship, for example the 'has phone' relationship between entities **PERSON** and **TELEPHONE** is a 1(p)-N(p), the encoding that an ERM tool has to provide would be:

```
rc1 [reltype->'normal', relseq->'x453',
     comp(person,telno,'hasPhone',0,'*',1)            -> telephone,
     comp(telephone,tpisof,'isOf',0,1,2)              -> person ].
```

In this relationship one component has entity **PERSON** with an attribute **TELNO** that can take zero to many identifiers of **TELEPHONE**. The other entity **TELEPHONE** has an attribute **TPISOF** that can take zero to one **PERSON** identifier. Note that the **RELTYPE** attribute of the referential constraint object **RC1** is '**NORMAL**' and its sequence number is '**X453**'.

Another binary relationship example, this time a 1(p)-N(t), is that a **PROJECT** must be led by a **LECTURER**. Its encoding is:

```
rc3 [reltype->'normal', relseq->'x683',
     comp(lecturer, prolead, 'leads', 0,'*',1)          -> project,
     comp(project,  leader,  'isLead',1,1,2)            -> lecturer ].
```

For the correctness of relationship instances we need check for two things. The first is that there should be no dangling references; that is an attribute is assigned a non-instantiated identifier and more so not being an instance-of the right class or structure. The second is its respect of the cardinality constraint mentioned in each of its component.

To check for dangling identifiers in a relationship structure a method, called **%RC_CONS_DANGLING_ID**, which takes two arguments namely a reference of a constraint and its type (e.g. 'normal' or 'weak') is called. The method reports any instance references which are not of the expected class or structure. To determine this each component of the relationship is read and from it three objects are used: the class (**?C**), attribute (**?A**) and artifact

being pointed to (**?Rt**). If there exists an instance-of **?C**, called **?I**, such that path expression **?I.?A** is not an instance-of **?Rt** then we have a dangling reference and possibly a type mismatch. The following is the generic method:

```
%rf_cons_dangling_id(?Rc, ?Type) :-
   ?Rc[reltype->?Type],
   ?Rc[comp(?C,?A,?_1,?_2,?_3,?_4)->?Rt], ?I:?C, ?I.?A=?Obj, not ?Obj:?Rt,
   write('-- rf cons dangling, problem with ')@_prolog,
   write(?A)@_prolog, write(' assigned ')@_prolog, writeln(?Obj)@_prolog,
   false.
```

Use of the procedure on referentially inconsistent data follows:

```
barbara:person     [fname->'barbara',     telno->{t123, 9999}].

?- %rf_cons_dangling_id(rc1,'normal').
-- rf cons dangling, problem with telno assigned 9999
No.
```

By assigning the lower and upper limits there are four possible cardinality settings in a **COMP** definition (see table 8.1). If for each occurrence we indicate if either a 'not null' or an 'at most one' constraint is required then each relationship cardinality is really a composition of these two. These conditions are sufficient and necessary.

Cardinality	Not Null	At most one
0:1		One & only
0:*		
1:1	Not null	One & only
1:*	Not null	

Table 8.1: Cardinality constraints composition of binary relationships

It quickly transpires that if the lower limit is "1" then a 'not null' has to be adhered to. Also if the upper limit is "1" then the 'at most one' constraint has to be maintained. Therefore to implement a "{0:1}" one needs to invoke 'one and only one' constraint while for "{1:1}" both 'not null' and 'one and only one" need to be respected.

In Flora-2 one way to implement the 'not null' constraint uses the negation as failure technique; i.e. if an attribute does not have a value assigned in an object then one can assume that the attribute value is null. The generic procedure to check if a relationship attribute is not null, called **%RF_CONS_NN**, is given here:

```
%rf_cons_nn(?Rc):-
 ?Rc[comp(?C,?A,?_1,1,?_3,?_4)->?_Rt],?I:?C,not ?I[?A->?_11],
    write('-- rf cons nn, problem with ')@_prolog,
    write(?I)@_prolog, write(' not assigned in ')@_prolog,
    writeln(?A)@_prolog,
    false.
```

The method finds the referential constraint of interest, i.e. through argument **?RC**, then if the component's lower limit (forth argument of **COMP**) is equal to '1' then any instance (**?I**) of class (**?C**) which does not have its attribute (**?A**) assigned to a value is breaking the constraint.

The method to check if an attribute is not being assigned more than one value is called **%RF_CONS_ONEONLY**. To detect attributes being assigned not more than one value we first sift components (i.e. **COMP**) with their upper limit set to '1', and then check if **?I.?A** takes two distinctive values.

```
%rf_cons_oneonly(?Rc):-
    ?Rc[comp(?C,?A,?_1,?_2,1,?_4)->?_Rt],
       ?I:?C,?I.?A=?V1, ?I.?A=?V2, ?V1!=?V2,
       write('-- rf cons oneonly, problem with ')@_prolog,
       write(?I)@_prolog, write(' assigned more than once in ')@_prolog,
       writeln(?A)@_prolog,
       false.
```

To run these three checks together a single method, called **%RF_CONS** of object **REFCONSTRAINT**, is provided. The method takes two arguments – a reference to a constraint and its type and then invokes the methods just described (actually without the false goal). The method is applicable on weak relationships too. The following is its code:

```
rfconstraint[%rf_cons(?Rc,?Type)]:-
    ?Rc[consname->?N], write('Ref cons name ')@_prolog, write(?N)@_prolog,
    write(' of type ')@_prolog, writeln(?Type)@_prolog,
    %rf_cons_seq(?Rc, ?Type).

%rf_cons_seq(?Rc,?Type):-   %rf_cons_dangling_id(?Rc,?Type), false.
%rf_cons_seq(?Rc,?_):-      %rf_cons_nn(?Rc),                false.
%rf_cons_seq(?Rc,?_):-      %rf_cons_oneonly(?Rc),           false.
```

The class of binary relationships with no attributes is distinctive for a practical reason. This class includes **1:1**, **1:N** and **M:N**, and recursive and weak style of relationships. Specifically this class of relationships is the one, and only, that ODMG ODL can implement with its relationship construct.

How can we encode relationship's attributes to a binary relationship? It is a well-known method for the cases of **1:N** and **1:1** one can move the relationship attributes to one of the entities. In the case of **1:N** the attributes are placed on to the many side entity. In the case of

the **1:1** it can be placed to either entity unless one component is in a total relationship in which case it is moved on to a total side.

For a binary many-to-many relationship with attributes there is a popular solution, which is adopted here, where a new class that will have a many-to-one referential constraint for each component class and an attribute for each relationship attribute. This new class is sometimes called the *resolving class*. This technique is found in text books such as El Masri [ELMAS10] and Teorey [TEORE05]. Its disadvantages are mainly two: the first is the creation of a class to implement a relationship and this is especially the case in binary relationships; and the second is that an *n*-ary relationship is spread over a number of binary relationships emanating from the resolving class.

Assume we have a **M(P)-N(P)** relationship, called **R1**, between classes **C1** and **C2** and the relationship attribute is called **A** and of type integer, then the corresponding encodings for classes and relationships would be:

```
C1:class     [ c1name{1:1} *=> string, c1_r1s{0:*} *=> C1C2r1 ].
C2:class     [ c2name{1:1} *=> string, c2_r1s{0:*} *=> C1C2r1 ].

C1C2r1:class [ c1s{1:1}    *=> C1, c2s{1:1}    *=> C2,
               A *=> integer].

r1_c1:refconstraint.
r1_c1[ consname-> 'r1 c1 desc'].
r1_c1[ reltype->  'normal', relseq->'x633',
               comp(C1, c1_r1s, 'r1 from c1', 0, '*', 1) -> C1C2r1,
               comp(C1C2r1, c1s, 'r1 to c1', 1, 1, 2) -> C1 ].

r1_c2:refconstraint.
r1_c2[ consname-> 'r1 c2 desc'].
r1_c2[ reltype->  'normal', relseq->'x633',
               comp(C2, c2_r1s, 'r1 from c2', 0, '*', 1) -> C1C2r1,
               comp(C1C2r1, c2s, 'r1 to c2', 1, 1, 2) -> C2 ].
```

It has to be noted that the participation constraints of a binary many-to-many relationship do not affect the participation constraint in the resolving class attributes, i.e. the **{1:1}** constraints in attributes **C1S** and **C2S** of class **C1C2R1** are always set so.

Another point to note is that our encoding is using attribute **RELSEQ** to relate the generated binary relationships with the original relationship. For example the two many-to-one relationships have the same value for **RELSEQ** i.e. **'X633'**.

What remains to be discussed is the primary key constraint of the relationship's resolving class. Previously in the primary key encoding we have shown that attributes forming part of

the key set can be of type reference to objects of a class. In our encoding here we use this to our advantage, which is ignoring the respective primary key set of component classes when defining the key set for the new class. In the case of binary many-to-many the key set would be composed of both references; i.e. in the case of **R1** above the key set is composed of **C1S** and **C2S** for **C1C2R1**.

```
pk1:pkconstraint.
pk1 [ consname  -> C1C2r1pk', classname -> C1C2r1,
      attrlist  -> {c1s, c2s} ].
```

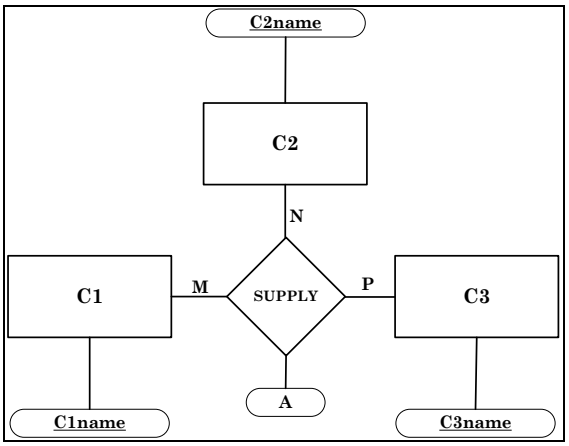


Figure 8.3 – Ternary relationship ERM

8.2.1.7 General *n* way Relationship templates (with *n* > 2)

For any *n*-ary relationship, with *n* greater than two, its encoding requires that a class is created to resolve the relationship. The new class will have a referential constraint to each of the participating entities. If the *n*-ary relationship has attributes then these are assimilated in the resolving class. For example a ternary relationship **R1** in figure 8.3 is a **N(P)-M(P)-P(P)** and is encoded into four classes with the forth (e.g. class **C1C2C3R1** in the following code) being the resolving class and it has a many-to-one binary relationship to each of the three participating classes. Note that each referential constraint related to the ternary relationship has the same value for attribute **RELSEQ**. This ensures one can recompose the *n* way relationship from its parts.

```
C1:class      [ c1name{1:1} *=> string, c1_r1s{0:*} *=> C1C2C3r1 ].
C2:class      [ c2name{1:1} *=> string, c2_r1s{0:*} *=> C1C2C3r1 ].
C3:class      [ c3name{1:1} *=> string, c3_r1s{0:*} *=> C1C2C3r1 ].

C1C2C3r1:class
              [ c1s{1:1} *=> C1, c2s{1:1} *=> C2, c3s{1:1} *=> C3,
                A *=> integer].
```

```
r1_c1:refconstraint.
r1_c1[ consname-> 'r1 c1 desc'].
r1_c1[ reltype->  'normal', relseq->'x8933',
               comp(C1, c1_r1s, 'r1 from c1', 0, '*', 1) -> C1C2C3r1,
               comp(C1C2C3r1, c1s, 'r1 to c1', 1, 1, 2) -> C1 ].

r1_c2:refconstraint.
r2_c2[ consname-> 'r1 c2 desc'].
r2_c2[ reltype->  'normal', relseq->'x8933',
               comp(C2, c2_r1s, 'r1 from c2', 0, '*', 1) -> C1C2C3r1,
               comp(C1C2C3r1, c2s, 'r1 to c2', 1, 1, 2) -> C2 ].

r1_c3:refconstraint.
r2_c3[ consname-> 'r1 c3 desc'].
r2_c3[ reltype->  'normal', relseq->'x8933',
               comp(C3, c3_r1s, 'r1 from c3', 0, '*', 1) -> C1C2C3r1,
               comp(C1C2C3r1, c3s, 'r1 to c3', 1, 1, 2) -> C3 ].
```

The primary key attribute set participants, for the resolving class, are dependent on the relationship's cardinalities constraint. Also, as presented in Chapter seven, primary key sets are a specialisation of a functional dependency constraint as a primary key set of attributes functionally determines all other attributes (i.e. not part of the primary key set) – in such a case the left hand side of a functional dependency is a candidate key. In the case of a binary N-M relationship, as just explained, the primary key set was composed of both references. Likewise for a ternary N-M-P the primary key set is a made of three references. The last row of table 8.2 depicts this. But the primary key set in other cardinality permutations doesn't need all three references composing the primary key set. Furthermore there is possibly that more than one candidate key set exists, for example in a 1-1-1 relationship, the resolving class can choose any primary key set from the following candidate keys: **{C1, C2}**, **{C2, C3}** and **{C1, C3}**.

Ternary Relationship (between C1, C2, C3)	**Functional dependencies**		
1-1-1	{C1, C2} → C3	{C1, C3} → C2	{C2, C3} → C1
1-1-N	{C1, C3} → C2	{C2, C3} → C1	
1-N-M	{C2, C3} → C1		
N-M-P	{C1, C2, C3} → ∅		

Table 8.2 – Functional dependencies present in ternary relationship by cardinalities

What is required is a general pattern that helps us enumerate all candidate keys for any *n*-ary relationship given its cardinalities and from which we can elect any one as primary. The remaining key sets are actually declared as candidate key constraints.

In the cases of N-M and N-M-P their primary key set is easy to derive, i.e. it includes all references, and there is only one pattern. It is easy to scale up for any *n*-ary relationship where the primary key set is made up of all entity references; i.e. *n* attributes.

In the case of an *n*-ary relationship where at least one of the cardinalities is a '1' then a reference to that class is placed on the right hand side of a functional dependence. On the left hand side of the dependence we append a reference of each of the remaining participants in the relationship. For example in a 1-N-M, between **C1**, **C2**, and **C3**, **C1** is a '1' component and is placed on the right of expression, whereas classes **C1** and **C2** are placed on the left hand side. Therefore the functional dependence built is **{C2, C3} → C1**. If a relationship has more than one '1' component a functional dependence is built for each. (Refer to table 8.2). This rule scales for *n*.

The following procedures and code snippets show how to, starting from an *n*-ary description of a relationship (e.g. figure 8.3), build the resolving class and enumerate its functional dependence and assert any one of these as the primary key and the rest as candidate keys. (For simplicity of implementation when we assert a primary key constraint, it is not asserted as a candidate key too). It needs to be remarked that during an actual exercise the invocation of this transformation has to be exposed to the designer; if incorrect or incomplete then the designer has to rectify the original input diagram.

It is reasonable to assume that in the case of an *n*-ary relationship the encoding requires more involved reading from the EERM tool. The encoding has a class for every participating entity together with their attributes and primary key. Something different is the reading of a class which is the resolving class for the *n*-ary relationship; the only attributes it should have are any relationship's attributes (i.e. **GRADE**). Also the *n*-ary relationship needs to be encoded as *n* binary relationships between the participating classes and the resolving class; the cardinalities and participation constraints on the side of each participating class are not changed. Every other side, i.e. from resolving to participating class, is assigned the same 'many and total'. Each of these binary relationships must have a common identifier in the **RELSEQ** attribute (e.g. **'X613'**). The following code shows one participating class (**STUDENT**) and the resolving class (named arbitrarily **SUL_TERNARY**) for figure 8.3.

```
student:class [sname{1:1} *=> string, sul_ternaries_s{0:*} *=> sul_ternary].
sul_ternary:class[t3student_id{1:1} *=> student, t3unit_id{1:1} *=> unit,
                  t3lecturer_id{1:1} *=> lecturer, grade{1:1} *=> integer ].

nmp_givenstud:refconstraint.
nmp_givenstud
  [reltype  -> 'normal',relseq->'x613',
   consname -> 'given student',
   comp( sul_ternary, t3student_id, 'from student',   1,   1, 1) -> student,
   comp( student, sul_ternaries, 'to sul ternary', 0, 1, 2) -> sul_ternary].
```

Once the ERM's *n*-ary relationship is read the framework needs to compute two procedures to implement the meaning of the *n*-ary relationship. The first builds all functional dependences in the resolving class to implement the relationship and the second is to assert a primary key and any other candidate keys to enforce the functional dependence just built.

The data characteristic in the framework of an *n*-ary is determined by a common value in the **RELSEQ** attribute across referential constraints instances – the variable **?RS** is used. For our example, there are three relationship instances with a common **RELSEQ** value; i.e. **'X613'**.

```
?- ?R:refconstraint[relseq->'x613'].
?R = nmp_givenlect
?R = nmp_givenstud
?R = nmp_givenunit
Yes.
```

Each of these constraint instances has two components and across all these is a component that points to their resolving class – the variable **?RESCLASS** is used. Therefore the class that is common across the constraint's components is the resolving class and to compute this one needs a 'for all' query – universal quantification. The following procedures compute this:

```
%rel_resolve_class(?Rs, ?ResClass):-
  ?R:refconstraint,
  ?R[relseq->?Rs,comp(?_11,?_12,?_13,?_14,?_15,?_16)->?ResClass],
  (not %rel_resolve_class_mid(?Rs, ?ResClass)),
  !.

    %rel_resolve_class_mid(?Rs, ?OutResClass):-
      ?R:refconstraint,
      ?R[relseq->?Rs, comp(?_21,?_22,?_23,?_24,?_25,?_26)->?_ResClass],
      (not %rel_resolve_class_in(?Rs, ?OutResClass, ?R)).

        %rel_resolve_class_in(?Rs, ?OutResClass, ?MidR):-
          ?R:refconstraint,
          ?R[relseq->?Rs, comp(?_31,?_32,?_33,?_34,?_35,?_36)->?ResClass],
          ?R=?MidR, ?ResClass=?OutResClass.

?- %rel_resolve_class('x613',?Rc).
    ?Rc = sul_ternary
Yes.
```

Once a resolving class of a relationship constraint is found then the computation of functional dependency needed to implement the *n*-ary relationship is done by the procedure **%REL_RESOLVE_FDS**. The procedure takes three input arguments and outputs a list of

dependencies (in **?FDELST**). The procedure starts by building two lists through the **COLLECTSET** aggregate function of Flora-2: the first being the list of participating classes in *n*-ary relationship (i.e. **?FDALLLST**); while the second is a list of classes that have at most a 'one' for an upper participation (i.e. **?FDRIGHTLST**). If the second set is empty then the procedure terminates and returns a list of fds. The following code explains:

```
%rel_resolve_fds(?Rs,?ResvClass,?FdSlst,?FdElst):-
  ?FdAlllst   = collectset{?C|?_R:refconstraint[relseq->?Rs,
                                comp(?ResvClass,?_2,?_3,?_4,?_5,?_6)->?C]},
  ?FdRightlst = collectset{?C|?_R:refconstraint[relseq->?Rs,
                                comp(?C,?_2,?_3,?_4,1,?_6)->?ResvClass]},
     if   (?FdRightlst=[])
     then (?FdElst=[?FdAlllst-[]])
     else (%rel_resolve_fds_gen(?FdAlllst,?FdRightlst,?FdSlst,?FdElst)).

     %rel_resolve_fds_gen(?FdAlllst,[?Rh|?FdRightlst],?FdSlst,?FdElst):-
        %loid_diff(?FdAlllst,[?Rh],[],?Lh),
        ?FdIlst = [?Lh-?Rh|?FdSlst],
        %rel_resolve_fds_gen(?FdAlllst,?FdRightlst,?FdIlst,?FdElst).
     %rel_resolve_fds_gen(?_FdAlllst,[],?FdElst,?FdElst).
```

If the ternary relationship is an N-M-P then there is only one functional dependency as the following invocation illustrates this. The response, bound to variable **?A**, says that **LECTURER**, **STUDENT** and **UNIT** identifiers determine all.

```
?-  %rel_resolve_fds('x613',sul_ternary,[],?A).

?A = [[lecturer, student, unit] - [] ]
```

If the second set is not empty (i.e. **?FDRIGHTLST**) then procedure **%REL_RESOLVE_FDS_GEN** is called iteratively for each element in this set. For each element the left hand side of the fd is built by extracting the difference from the set **?FDALLLST** by using the procedure **%LOID_DIFF**. If the ternary relationship is 1-1-1 then there is a list of fds and the following invocation illustrates this. The response, bound to variable **?A**, says there are three fds and the first says that student and lecturer identifiers determine the unit's identifier.

```
?-  %rel_resolve_fds('x613',sul_ternary,[],?A).

?A = [[student, lecturer] - unit, [unit, lecturer] - student, [unit,
student] - lecturer ]
```

Given a non-empty set of fds one needs to convert these into constraints – the procedure **%REL_RESOLVE_KEYS** takes care of this. The first fd is arbitrarily converted into a primary key and any remaining fds into candidate key constraints. Because the framework considers a constraint as an object it is imperative that the generated fds are asserted as objects in the framework's knowledge base. To generate objects from rules Flora-2 offers *reification*

mechanism based on Yang and Kifer's work [YANGG03]; the rules with un-instantiated placeholders are assigned as values to methods of the generic constraint object (and enclosed within **${** and **}** tokens) and when required to instantiate a constraint the place holders are filled and the rule fired to create a new object. Other than reification we also need a predicate that when instantiated returns a logical identifier which is new and unique; the directive is called **NEWOID{}**. The following code snippets illustrate how constraints are generated.

The generic object **PKCONSTRAINT**, and similarly for candidate key object, have two methods and each of which take two arguments and whose value is a reified rule. For example when method **CREATE_PK_INSTANCE** is invoked with a new identifier and a class identifier the rule is bound with the arguments and ready to fire; that is create a new **PKCONSTRAINT** instance with the new identifier and it is **CLASSNAME** attribute assigned the class identifier provided as an argument. The second method, **CREATE_PK_INSTANCE_ATTR**, is responsible to assign attributes that make up the primary key constraint through the attribute **ATTRLIST**.

Once the rules are in place, then the procedure **%REL_RESOLVE_KEYS** drives the instantiation by taking the first fd from the list and passing it to generate the primary key at **%REL_RESOLVE_KEYS_PK** which generates a new identifier, instantiates the rule to fire, and finally fires the rule with **INSERTRULE{…}** directive. A similar treatment is applied for candidate keys but with the difference that procedure iterates until all fds have an object instantiation.

```
pkconstraint[create_pk_instance(?Loid, ?Cn) ->
     ${ ( ?Loid:pkconstraint[classname->?Cn] :- true) } ].
pkconstraint[create_pk_instance_attr(?Loid, ?Attr) ->
     ${ ( ?Loid[attrlist->?Attr] :- true ) } ].
ckconstraint[create_ck_instance(?Loid, ?Cn)  ->
     ${ ( ?Loid:ckconstraint[classname->?Cn] :- true) } ].
ckconstraint[create_ck_instance_attr(?Loid, ?Attr) ->
     ${ ( ?Loid[attrlist->?Attr] :- true ) } ].

%rel_resolve_keys(?_Rs,?ResvClass,[?H|?RFdlst]):-
       %rel_resolve_keys_pk(?ResvClass,?H),
       %rel_resolve_candkeys(?ResvClass,?RFdlst).
%rel_resolve_keys(?_Rs,?_ResvClass,[]).


    %rel_resolve_keys_pk(?Cn,?Alst-?_Rh):-
          newoid{?Loid},
          pkconstraint[create_pk_instance(?Loid,?Cn) -> ?Rules1],
          insertrule { ?Rules1 },
          %rel_resolve_keys_pk_attr(?Loid,?Alst).
```

```
        %rel_resolve_keys_pk_attr(?Loid,[?H|?Alst]):-
              pkconstraint[create_pk_instance_attr(?Loid,?H) -> ?Rules1],
              insertrule { ?Rules1 },
              %rel_resolve_keys_pk_attr(?Loid,?Alst).
        %rel_resolve_keys_pk_attr(?_Loid,[]).


    %rel_resolve_candkeys(?ResvClass,[?H-?_Rh|?RFdlst]):-
          newoid{?Loid},
          ckconstraint[create_ck_instance(?Loid,?ResvClass) -> ?Rules1],
          insertrule { ?Rules1 },
          %rel_resolve_candkeys_attr(?Loid,?H),
          %rel_resolve_candkeys(?ResvClass,?RFdlst).
    %rel_resolve_candkeys(?_ResvClass,[]).

          %rel_resolve_candkeys_attr(?Loid,[?H|?Alst]):-
              ckconstraint[create_ck_instance_attr(?Loid,?H) -> ?Rules1],
              insertrule { ?Rules1 },
              %rel_resolve_candkeys_attr(?Loid,?Alst).
          %rel_resolve_candkeys_attr(?_Loid,[]).
```

The three top level procedures just described are wrapped within a generic object procedure – namely **RFCONSTRAINT[%FD_GENERATE(ARG1, ARG2, ARG3)]** which follows. Also a query to check the primary and candidate keys generated is appended to the code below. (Note: identifier starting with **'_#'** are identifiers generated by calling **NEWOID{…}**).

```
rfconstraint[%fd_generate(?Rs,?ResvClass,?Fdlst)]:-
    %rel_resolve_class(?Rs,?ResvClass),
    %rel_resolve_fds(?Rs,?ResvClass,[],?Fdlst),
    %rel_resolve_keys(?Rs,?ResvClass,?Fdlst).

?- rfconstraint[%fd_generate('x613',?ResvClass,?Fdlst)].
?ResvClass = sul_ternary
?Fdlst = [[student, lecturer] - unit, [unit, lecturer] - student, [unit,
student] - lecturer]

?- ?Pk:pkconstraint[attrlist->?L].

?Pk = _#'10318  ?L = lecturer
?Pk = _#'10318  ?L = student

?- ?Ck:ckconstraint[attrlist->?L].

?Ck = _#'10319  ?L = lecturer
?Ck = _#'10319  ?L = unit
?Ck = _#'10320  ?L = student
?Ck = _#'10320  ?L = unit
```

For *n*-ary relationships we have used the ternary as an example. In fact the structures, procedures and output are independent of degree 3 and are really applicable to $n \geq 2$.

8.2.1.8 Other relationship templates

In the previous sections a number of relationships have been encoded into structures and their semantics enforced through integrity constraints. Some of these relationships needed a transformation into other structures and additional constraints to enforce them. In summary static constraints and structural artefacts are sufficient to encode them.

Other relationships, mentioned in an chapter two like aggregation, require more than structures and static constraints to implement them in full. Specifically aggregation constructs need transitional constraints; these are typically implemented with database triggers. (In fact an earlier paper by the author [VELLA97] aggregation design depended on their presence). In ODMG ODL and OQL no mention of triggers is forthcoming. Consequently it is best left as a post mapping exercise rather than part of the database schema generated here; in fact both in this research and in our contribution to ICOODB 2009 [VELLA09].

8.3 Summary

This chapter builds a framework in which an object-oriented database is defined and maintained in terms of its data model. The language used for implementation is Flora-2 and much of the programming is declarative. These procedures are attached to framework objects.

The "meta" structures in our framework include: classes, weak entities, composite object parts, attributes (including functional ones), domains, relationships, and constraints (e.g. primary key, referential, check, not null, and functional dependency). Each artefact is supported, safeguarded, and enforced through rules and methods defined for each of them. In all the programming mixing of attribute values and their data type signatures is extensive.

The framework also offers methods to aid database designers in some of their activities. For example there is a method that for an *n*-ary relationship instance (with n>2) is capable of converting it into the equivalent binary relationships, a resolving class, and constraints.

The framework is now able to accept a variety of schema designs. In fact in the next chapter we see how we can convert an EERM diagram encoding into our framework artefacts.

Chapter 9

Translating EERM into an ODL Schema

9 Translating an EERM into an ODL Schema

The chapter starts by presenting an encoding of an EERM design into our framework meta structures. That is the entities, relationships, and constraints encoded in an EERM diagram are asserted as facts in the framework. Together with these design facts the framework has a number of rules that encode and check properties that an object database must satisfy.

Once the EERM diagram is encoded as facts it is mapped into a sequence of ODMG ODL constructs. These constructs, albeit in an ODMG ODL dialect, are directly used by EyeDB to create its own object schema. EyeDB constructs used are classes with attributes and relationships, ISA assertions, and integrity constraints (e.g. primary key and not null). Through the mapping other constructs, not found in the original EERM, are introduced to enable a wider coverage of our mapping to ODMG data model; for example in non-binary relationships.

Other than the logical encoding of EERM diagrams in Flora-2, we also present a straightforward utility that redraws an EERM in a graphical form as new constructs may be added. It is useful to have a visualisation of the EERM model encoded in our framework. This is similar to the actual EERM but some differences have been introduced. For example an EERM *n*-ary relationship is converted into *n* binary relationships and an addition of a resolving class.

There is a point worth emphasising here is that specification made in the EERM diagram are to remain present with different functionalities applied to the object-database; for example an object query language is to make use of these EERM relationship artefacts.

9.1 The Problem Definition

Drawing a graph is a non-trivial computational task as any projection, even if correctly representing an underlying mathematically defined graph, can give different clues to its reader. A number of software packages that take a graph specification, spruced up with drawing attributes, and then output a diagram into raster and vector formats exist. Vector formats are useful as they allow editing of a generated diagram. Most packages offer command-line and graphical-user interfaces.

Two packages that have a good following, active revision process and proper documentation are GraphViz { WWW.GRAPHVIZ.ORG } and yEd { WWW.YWORKS.COM }. For this work GraphViz, a product originating from AT&T Labs Research, has been chosen mainly for its reliability, availability of packages and support, and finally because it is able to meet this project's requirements.

The idea is to faithfully convert the conceptual design elements into a diagram's artefacts. The mapping has to have a high-level completeness and correctness. Completeness is in the sense all elements of the EERM must be present in the output (or accounted for), and correctness implies that any artefact's meaning is not changed by the mapping.

The main input for the diagraming tool are nodes and edges. Both take a variety of properties that define their structure, name and caption, and formatting style. The package also allows for generic instructions that cover the whole diagram. In our mapping the nodes will represent the entities, weak entities, and all types of attributes. Edges will represent relationships: the binary ones are undirected edges but are adorned with role name, participation and cardinality constraints; the *ISA* ones use directed edges.

Since the framework is repository for an EERM graph and a number of checks and constraints are enabled then it is a rather straightforward procedure to implement. There are two issues that make the process slightly tricky. The first concerns inheritance: in most representations one shows the additional properties rather than all properties including those inherited. In F-logic, if one asks for an object's properties then one gets both inherited and non-inherited attributes. The second concerns binary relationships: the double ended nature of a binary relationship makes every relationship be represented twice; albeit having different perspectives.

9.1.1 General Procedure

An object **`SCHEMA`** method, called **`%VIZ_SCHEMA`**, specifies the schema encoding drawing. Prior to calling **`%VIZ_SCHEMA`** one has to confirm that all integrity constraints entrenched in the framework are adhered to otherwise this mapping is vitiated.

A high-level description of the method follows: the first part takes care of listing the header part of the specification file, and the second part enumerates the entities, ISA relationship,

attributes, and relationships and converts them to into graphical equivalents. Thirdly the edges are introduced that link the nodes just produced. In the following description of the translation the procedures have been stripped to their bare essentials for readability in the text.

The procedure **%VIZ_HEADER** is called by **%VIZ_SCHEMA**. It starts by writing basic details in a remark, for example schema name, and then sets graphical properties of the diagram. One of these is **DIGRAPH** directive which asks Dot, one of GraphViz's engines, to render a directed graph; this is required as the *ISA* relationship is directed. Also the directive names the diagram and opens the specification list with a brace token.

```
%viz_header :-
  write  ('    //  schema name & version: ')@_prolog,
  write  (schema.schemaname)@_prolog,
  writeln('digraph FlogicEncode {')@_prolog.
```

The procedure **%VIZ_FOOTER**, called at the end of **%VIZ_SCHEMA** just closes in the input specification and therefore is called at the end of the process.

The generated code looks as follows (the ellipsis, i.e. …, indicates further text has been omitted for brevity).

```
/* flogic schema for visualisation */
    ##  schema name & version: AfterScott (alpha)
digraph FlogicSchema {
ranksep=1.25;
…}
```

9.1.2 Entities, and Weak Entities

The main nodes of the diagram are classes and weak entities and both of which are instances of **CLASS** and **STRUCTURE** in the framework. To print these into a diagram specification two procedures are called in sequence. The first is called **%VIZ_CLASS** and the second is **%VIZ_STRUCTURE**.

%VIZ_CLASS unifies with a list of classes, i.e. using view **ALLCLASSES.CLASSLIST**, and recursively calls method **%VIZ_CLASS_ENTRY** with the **CLASS** instances list, each method invocation uses a class instance at a time to print it out as a Dot specification. The output includes the class name and its shape attributes (e.g. **SHAPE=BOX**). Dot, unless otherwise instructed, includes the class name as the node's caption within the box drawn.

```
allclasses[classlist->?L]  :- ?L = collectset{ ?C | ?C:class }.

%viz_class :-
  %viz_class_entry(allclasses.classlist).

  %viz_class_entry([?C|?Clrest]):-
    write(?C)@_prolog,
    writeln('[shape=box];')@_prolog,
    %viz_class_entry(?Clrest).
  %viz_class_entry([]):- true.
```

The next method to be called is **%VIZ_STRUCTURE** and is very similar to **%VIZ_CLASS**. The differences being: first it uses a list of **STRUCTURES**; second each **STRUCTURE** is checked if it is involved in a **WEAK** relationship instance (i.e. **?_RF:REFCONSTRAINT [RELTYPE->'WEAK', COMP(?S,?_,?_,?_,?_,?_)->?_]**); and thirdly the shape is different in that a weak entity is represented with a double edged box (i.e. **SHAPE=BOX, PERIPHERIES=2**).

```
allstructures[structlist->?L]  :- ?L = collectset{ ?S | ?S:structure }.

 %viz_structure :-
     %viz_structure_entry(allstructures.structlist).
 %viz_structure :- true.

    %viz_structure_entry([?S|?Sl]):-
      if ( ?_Rf:refconstraint[reltype->'weak',comp(?S,?_,?_,?_,?_,?_)->?_] )
      then ( write(?S)@_prolog,
             writeln('[shape=box, peripheries=2];')@_prolog ),
      %viz_structure_entry(?Sl).
    %viz_structure_entry([]):- true.
```

The generated code looks as follows:

```
…
/* classes */
    course[shape=box,height=.75,width=1.5];
…
/* structures - weak entities */
    address[shape=box,height=.75,width=1.4,peripheries=2];
…
```

9.1.3 ISA Relationship

The next method call from **%VIZ_SCHEMA** is to **%VIZ_ISA** that draws all *ISA* relationships from a class to its direct descendent. The latter calls procedure **%VIZ_ISA_ENTRY** twice; the first time it is called with the **CLASSES** list and the second time with the **STRUCTURES** list. In **%VIZ_ISA_ENTRY** a list of *ISA* relationship objects whose parent object is the head of the list passed as an argument is created (i.e. **COLLECTSET { ?I| ?I:ISAPROPERTY, ?I[PARENTCLASS->?C] }**). The **CLASS** object and its list of *ISA* instances are passed to method **%VIZ_ISA_ENTRY_DETAIL**. When invoked this procedure extracts data of one *ISA* relationship instance at a time and starts the drawing process. (Actually there is a constraint in the framework that enforces an upper limit of one *ISA* relationship).

We recall that the *ISA* relationship drawing requires a circle to annotate its details (i.e. disjoint versus over-lapping, and total versus partial) and two types of edges. The first type of edge is from the parent class to the circle, and the second type of edge is from the circle to each descendent of the parent class. The line drawing from the circle to the descendants is done by procedure **%VIZ_ISA_ENTRY_DETAIL_DESC**. The edges have annotations that were extracted from the relative *ISA* relationship instance.

Each procedure mentioned above is recursive, except for **%VIZ_ISA**. Some code follows and within it one notes that **FWRITE** procedure does not really exist but makes reading easier.

```
%viz_isa_entry([?C|?Rl]):-
  ?Isa=collectset{?I|?I:isaproperty, ?I[parentclass->?C]},
  %viz_isa_entry_detail(?C,?Isa),
  %viz_isa_entry(?Rl).
%viz_isa_entry([]):- true.

  %viz_isa_entry_detail(?C,[?I|?Rest]):-
    ?I[disj_over_flag->?doF, totl_part_flag-> ?tpF, sema_pred_flag->?spFl],
    fwrite(?I, '[shape=circle, label="', ?doF, '"];'),
    fwrite(?C, '->', ?I,'[label="',?tpF,'", dir="both",',
           'arrowhead=none, arrowtail="vee", color="gray:gray"];'),
    ?Ldesc=collectset{?D|?I[isaclass->?D]},
    %viz_isa_entry_detail_desc(?I,?Ldesc,?spF),
    %viz_isa_entry_detail(?C,?Rest).
  %viz_isa_entry_detail(?_C,[]).

    %viz_isa_entry_detail_desc(?I,[?D|?Rest],?spF):-
      fwrite(?I,'->',?D, '[label="', ?spF, '", arrowhead=none];'),
      %viz_isa_entry_detail_desc(?I,?Rest,?spF).
    %viz_isa_entry_detail_desc(?_I,[],?_spFlag).
```

The generated code looks as follows. Node **ISAPERSON** is representing the *ISA* relationship instance descriptor (i.e. denoted in the EERM as a circle) and is captioned with 'disjoint', node **PERSON** is a parent class and nodes **LECTURER** and **STUDENT** are its sub-classes connected through **ISAPERSON** with two types of edges. The first is from **PERSON** to **ISAPERSON** and is captioned with 'partial', and the second is from **ISAPERSON** to a sub-class and each of them is captioned with 'semantic'.

```
…
isaperson
  [shape=circle, height=.5, width=.5,
   fontsize=12, fixedsize=true, label="disjoint"];
person -> isaperson
  [label="partial", fontsize=10, dir="both", arrowhead=none,
   arrowtail="vee", color="gray:gray", weight=10];
isaperson -> lecturer
  [label="semantic", arrowhead=none, color="gray:gray", fontsize=10];
isaperson -> student
  [label="semantic", arrowhead=none, color="gray:gray", fontsize=10];
…
```

Since asserting *ISA* facts in F-logic implies inference of its closure on program evaluation one needs, for drawing purposes, to weed off this closure. In procedure **%VIZ_ISA** we choose to enumerate *ISA* facts through the specific relationship encoding rather than creating a list through the pattern **?C:CLASS, ?CD:CLASS, ?CD::?C**.

9.1.4 Binary Relationships

The framework has all EERM relationships encoded as binary and all referential constraints manifest themselves as of three types: either normal, or weak, or structure (for building composite objects). The former two are mapped in this section's code. The called procedure **%VIZ_RELATIONSHIP** invokes **%VIZ_REL_NORMAL** twice, first with all **CLASS** instances and then with all **STRUCTURE** instances. The method **%VIZ_REL_NORMAL** is a recursive procedure over the list of object references in its argument.

On each invocation of **%VIZ_REL_NORMAL** the method builds a list of relationships that are involved with the current object reference being evaluated. The respective filter for normal relationships follows: **?R:REFCONSTRAINT, ?R[RELTYPE -> 'NORMAL'], ?R[COMP(?CL,?_,?_,?_,?_,1) -> ?_]**. Also note the "**1**" value in the last argument of a relationship component that ensure each relationship instance, being of double ended nature, only appears once as this argument is a serial number. The method then calls **%VIZ_REL_NORMAL_EDGE** with two arguments: the first being the current **CLASS** instance (or **STRUCTURE** instance) and the second is a list of relationships just computed.

```
%viz_relationship :-
   %viz_rel_normal(allclasses.classlist),
   %viz_rel_normal(allstructures.structlist).

   %viz_rel_normal([?Cl|?Rl])       :-
      ?Rel=collectset{?R|?R:refconstraint,?R[reltype->'normal'],
                        ?R[comp(?Cl,?_,?_,?_,?_,1)->?_]},
      %viz_rel_normal_edge(?Cl,?Rel),
      %viz_rel_normal(?Rl).
   %viz_rel_normal([])              :- true.

      %viz_rel_normal_edge(?Cleft,[?R|?Rest])   :-
          ?R[comp(?Cleft ,?_,?Label,?Lmin,?Lmax,1)->?_],      // left side
          %viz_rel_edge_lu(?Lmin,?Lmax,?Aleft),
          ?R[comp(?Cright,?_,?_,    ?Rmin,?Rmax,2)->?Cleft], // right side
          %viz_rel_edge_lu(?Rmin,?Rmax,?Aright),
          // drawing
          write(?Cleft)@_prolog, write('->')@_prolog, write(?Cright)@_p,
          write('[label="')@_prolog, write(?Label)@_prolog,
          write('",dir="both"')@_prolog,
          write(',arrowtail=')@_prolog,write(?Aleft)@_prolog,
          write(',arrowhead=')@_prolog,write(?Aright)@_prolog,
          writeln('];')@_prolog,
          %viz_rel_normal_edge(?Cleft,?Rest).
```

```
%viz_rel_normal_edge(?_,[]):- true.

 %viz_rel_edge_lu(0,1,  ?E) :- ?E = '"teeodot"',!.
 %viz_rel_edge_lu(0,'*',?E) :- ?E = '"crowodot"',!.
 %viz_rel_edge_lu(0,?_,?E)  :- ?E = '"crowodot"',!.

 %viz_rel_edge_lu(1,1,  ?E) :- ?E = '"teetee"',!.
 %viz_rel_edge_lu(1,'*',?E) :- ?E = '"crowtee"',!.
 %viz_rel_edge_lu(1,?_,?E)  :- ?E = '"crowtee"',!.
```

The method **%VIZ_REL_NORMAL_EDGE** is recursive too and needs to be called for each element of the relationship instances list; i.e. second argument. For each instance the method needs to draw an edge from the first component to the second of the current relationship instance. (These have been arbitrarily named as left and right in the code). The matching with a relationship instance allows the extraction of other data: namely lower and upper limits for each component and a caption for the edge drawn. For the left side these are called **?LMIN**, **?LMAX**, and **?LABEL**. There are four possible types of edges to draw for each end of an edge and calling method **%VIZ_REL_EDGE_LU** with the respective lower and upper limits returns the type of edge for Dot to draw. The pictograms selected from Dot's library are depicted in figure 9.1 and are assigned to edge properties **ARROWHEAD** and **ARROWTAIL** for the edge being drawn.

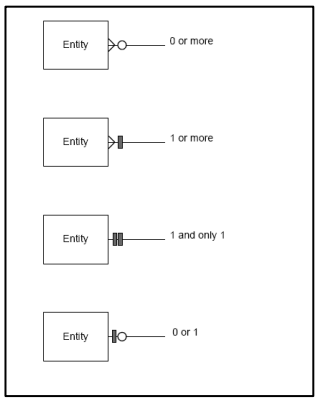


Figure 9.1 – Depicting cardinality and participation constraints of a relationship.

The generated code looks as follows:

```
…
/* relationships */
    course->student[label="enrolled on",
                    dir="both", arrowtail="crowodot", arrowhead="teetee"];
…
```

The procedure to draw a weak relationship called **%VIZ_WEAK_RELATIONSHIP** is almost identical to the previous. The only difference is in the query to collect weak relationships in a list. The rest re-uses the procedures used for normal relationships.

```
%viz_weak_relationship :-
   %viz_rel_weak(allclasses.classlist),
   %viz_rel_weak(allstructures.structlist).

   %viz_rel_weak([?Cl|?Rl])       :-
      ?Rel=collectset{?R|?R:refconstraint,?R[reltype->'weak'],
                        ?R[comp(?Cl,?_,?_,?_,?_,1)->?_]},
      %viz_rel_normal_edge(?Cl,?Rel),
      %viz_rel_weak(?Rl).
   %viz_rel_weak([])              :- true.
```

9.1.5 Entity Attributes

There are four types of attributes we need to depict; namely single-simple, set-simple, single-structure and set-structure. These attributes in an EERM diagram are found in entities, weak entities, and relationships. In our framework there is a reduction of *n*-ary relationships into a set of binary ones and relationships are disarmed of any attributes. This leaves only the former two for drawing. Furthermore in our framework a structured attribute forming part of an entity is implemented with a binary relation (of type structure - **RELTYPE->'STRUCTURE'**).

To draw attributes we partition these into two main groups. The first is earmarked to generate the drawing for a structured attribute and its parts that is encoded in the framework as a structure instance and related with a relationship instance-of type structure (i.e. **RELTYPE->'STRUCTURE'**). This procedure is called **%VIZ_STRUCT_RELATIONSHIP**. The second group takes care of the other attributes, specifically attributes for entities and weak entities that are not structured attributes supported by the first procedure. This procedure is called **%VIZ_ATTRIBUTE**.

On each invocation of **%VIZ_ATTRIBUTE** it calls another four procedures to take care of single-simple, set-simple, single-structure, and set-structure attributes. These are named **%VIZ_ATTR_ELLIPSE**, **%VIZ_ATTR_DOUBLE_ELLIPSE**, **%VIZ_ATTR_STRUCTURE**, and **%VIZ_ATTR_DOUBLE_STRUCTURE** respectively. These procedures have a common thread and differ mainly in their rendering of shapes.

The procedure **%VIZ_ATTR_ELLIPSE** calls **%VIZ_ATTR_ALL_ELLIPSES** twice first with a list of classes and then a list of structures. In **%VIZ_ATTR_ALL_ELLIPSES** the head of the list, e.g. a **CLASS** instance, is used to create a new list of **ATTRIBUTES** attached to this **CLASS** instance. As stated previously we expect to have only **ATTRIBUTES** that are defined in this **CLASS** instance and exclude any inherited ones as these are shown in their defining **CLASS** instance.

To compute this list the framework has a number of views (i.e. rules) that derive these attributes. For simple and single attributes the following views are created:

```
?C(?A,'attr'):gviz[attrtype -> ?Dt, attrcard -> 'single'] :-
    ?C:class[?A{?_:?Up}*=>?Dt],
    ?Up=1, (?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
    if ( ?C::?Sc,?Sc:class[?A{?_:?Up}*=>?Dt] )
    then ( false )
    else ( true  ).

?S(?A,'attr'):gviz[attrtype -> ?Dt, attrcard -> 'single'] :-
    ?S:structure[?A{?Low:?Up}*=>?Dt],
    ?Up=1, (?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
    if ( ?S::?Sc,?Sc:structure[?A{?Low:?Up}*=>?Dt] )
    then ( false )
    else ( true  ).
```

The body of the first rule looks for **CLASS** instance attributes whose signature matches the following conditions: upper limit is one, and return data type is a basic domain. Furthermore we are interested in attributes defined in current class and not inherited; this is taken care of by the pattern **?C::?SC,?SC:CLASS[?A{?_:?UP}*=>?DT]** that is if instantiated it indicates that the method is defined in an ancestor **CLASS**. If the body is true the head creates a new object as an instance-of **GVIZ** with an identifier of **?C(?A,'ATTR')**. The object created also has two properties assigned: namely **ATTRTYPE** and **ATTRCARD**. The second rule is very similar to the first except its scope is over instances of **STRUCTURES**.

The procedure **%VIZ_ATTR_ALL_ELLIPSE** collects for the currently instantiated **CLASS** instance all of its attributes that have '**ATTR**' in part of their identifier in the **GVIZ** view. The current class instance and this list of attributes are passed to procedure **%VIZ_ATTR_ONE_ELLIPSE** to draw an oval and an edge for each attribute in the list. The procedures outputs two directives to Dot: the first for the oval as a node and the second as an edge from the class instance to the attribute (i.e. the oval). This procedure has to figure out if the current attribute queued for printing is actually part of the class instance primary key set (i.e. **?_PK:PKCONSTRAINT[CLASSNAME->?E, ATTRLIST->?A]**) because in which case the oval is printed in bold rather than in plain style. The relative procedures declarative code follows:

```
%viz_attr_all_ellipses([?E|?Re]):-
  ?Cssa = collectset{ ?A | ?E(?A,'attr'):gviz },
  %viz_attr_one_ellipse(?E,?Cssa),
  %viz_attr_all_ellipses(?Re).
%viz_attr_all_ellipses([]):-  true.

  %viz_attr_one_ellipse(?E,[?A|?Ra]):-
    write(?E)@_prolog, write(?A)@_prolog,
```

```
    write('[shape=ellipse, label=')@_prolog,
    write(?A)@_prolog,
    if  (?_Pk:pkconstraint[classname->?E, attrlist->?A])
    then ( writeln(',style=bold];')@_prolog )
    else ( writeln('];')@_prolog ),
    write(?E)@_prolog, write('->')@_prolog,
    write(?E)@_prolog, write(?A)@_prolog, writeln('[arrowhead=none]; ')@_p,
    %viz_attr_one_ellipse(?E,?Ra).
  %viz_attr_one_ellipse(?_,[]):-  true.
```

Typical output follows:

```
…
/* attributes – single and simple (and part of pk)  */
coursecname[shape=ellipse,width=.75,height=.5,label=cname,style=bold];
course->coursecname[arrowhead=none];
…
```

The procedure **%VIZ_ATTR_DOUBLE_ELLIPSE** takes care of set-simple attributes and is very similar to the previous one just described. The first differences is the upper bound of an attribute (i.e. **?UP**) is bound to a value not equal to one; for example when an upper bound is equal to '*****'. The second difference concerns the identifier of the view created to relate with the relevant set attributes associated with a class instance is '**SETATTR**' rather than '**ATTR**". Thirdly the shape is an ellipse but has a double border that is specified with a Dot qualifier called "**PERIPHERALS=2**".

The procedure to cater for single-structure attributes but not involved in a relationship instance-of type structure is called **%VIZ_ATTR_STRUCTURE**. The views associated with this procedure checks for **CLASS**es and **STRUCTURE**s attribute signature whose upper limit (i.e. **?UP**) is set one and its return data type is not a basic domain. Furthermore the body of the rule checks for two cases that inhibit firing the rule head. The first being that there is no relationship instance that uses this **CLASS** and **ATTRIBUTE**, and secondly that there is no ancestral class that has this attribute signature already defined. Once fired the following rules will populate the views **?C(?A,'ATTR')** and **?S(?A,'ATTR')**.

```
?C(?A,'structattr'):gviz[ attrtype->?Dt, attrcard->'single'] :-
    ?C:class[?A{?Low:?Up}*=>?Dt],
    ?Up=1, not(?Dt=?C;?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
    if   (not ?_:refconstraint[comp(?C,?A,?_,?_,?_,?_) -> ?_])
    then ( if  (?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt])
           then ( false ) )
    else ( false ).

?S(?A,'structattr'):gviz[ attrtype -> ?Dt, attrcard->'single'] :-
    ?S:structure[?A{?Low:?Up}*=>?Dt],
    ?Up=1, not(?Dt=?S;?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
    if   (not ?_:refconstraint[comp(?S,?_,?_,?_,?_,?_) -> ?_])
    then ( if   (?S::?Sc,?Sc:structure[?A{?Low:?Up}*=>?Dt])
           then ( false ) )
    else ( false ).
```

For each **CLASS** and **STRUCTURE** instance the procedure generates a list of **ATTRIBUTES** that need to be attached to them and a call is made to procedure **%VIZ_ATTR_ONE_STRUCTURE** with these two as arguments. In the later procedure it first builds a node for the attribute whose shape is an ellipse and named as a concatenation of class and attribute names, and then builds an edge from the class instance to the node just defined (i.e. class and method names concatenation). The procedure calls itself until all attributes attached to the **CLASS** instance have node and an edge built. The code follows:

```
%viz_attr_all_structures([?E|?Re]):-
  ?Cssa = collectset{ ?A | ?E(?A,'structattr'):gviz },
  %viz_attr_one_structure(?E,?Cssa),
  %viz_attr_all_structures(?Re).
%viz_attr_all_structures([]):- true.

  %viz_attr_one_structure(?E,[?A|?Ra]):-
    write(?E)@_prolog, write(?A)@_prolog,
    write('[shape=ellipse,,label=')@_prolog, write(?A)@_prolog,
    if   (?_Pk:pkconstraint[classname->?E, attrlist->?A])
    then ( writeln(',style=bold];')@_prolog )
    else ( writeln('];')@_prolog ),
    write(?E)@_prolog, write('->')@_prolog,
    write(?E)@_prolog, write(?A)@_prolog,
    writeln('[arrowhead=none]; ')@_prolog,
    %viz_attr_one_structure(?E,?Ra).
  %viz_attr_one_structure(?_,[]):- true.
```

The procedure **%VIZ_ATTR_DOUBLE_STRUCTURE** is similar to the previous but applies a double perimeter if the upper limit is not equal to one. The previous procedure **%VIZ_ATTR_DOUBLE_ELLIPSE** is a basis for it.

What remains to describe is the composite structure attributes that are also constrained by a relationship instance. The procedure **%VIZ_STRUCT_RELATIONSHIP** ensures that a composite object is depicted in the diagram. The procedure in turn calls **%VIZ_STRUCT_RELATIONSHIP_COMP** twice with a list of all classes and then with list of all structures. In the latter procedure, which recurses on the list of identifiers passed as argument, it computes a list of referential constraint instances that are first of type structure (i.e. **RELTYPE->'STRUCTURE'**), and second have a relationship component that maps from the procedure's argument to a structure instance (i.e. **COMP(?C1, ?_, ?_, ?_, ?_, ?_) -> ?S**). For this list of referential identifiers and relative class instances the procedure **%VIZ_STRUCT_RELATIONSHIP_COMP_EDGE** is called to build the required nodes and edge. The procedure does three things: first it extracts details from the relative referential constraint instance, second it introduces the structure instance as an ellipse node (rather than

a rectangle) with the right number of perimeters (e.g. double for set), and thirdly draws an edge from object just created denoting its class instance. The components of the structure are taken care of by the other procedures described earlier. The following code applies:

```
%viz_struct_relationship :-
  %viz_struct_relationship_comp(allclasses.classlist),
  %viz_struct_relationship_comp(allstructures.structlist).

  %viz_struct_relationship_comp([?Cl|?Rl]) :-
    ?Rel=collectset{?R|?R:refconstraint, ?R[reltype->'structure',
                      comp(?Cl,?_,?_,?_,?_,?_)->?S],?S:structure},
    %viz_struct_relationship_comp_edge(?Cl,?Rel),
    %viz_struct_relationship_comp(?Rl).
  %viz_struct_relationship_comp([]) :- true.

     %viz_struct_relationship_comp_edge(?Cleft,[?R|?Rest]) :-
        ?R:refconstraint[reltype->'structure',
                         comp(?Cleft,?A,?_,?_,?Up,?_)->?S], ?S:structure,
        fwrite(?S,'[shape=ellipse, label=',?A),
        if   (?Up=1)
        then (writeln(']; ')@_prolog)
        else (writeln(',peripheries=2]; ')@_prolog),
        fwrite(?Cleft, '->', ?S),
        writeln('[arrowhead=none]; ')@_prolog,
        %viz_struct_relationship_comp_edge(?Cleft,?Rest).
      %viz_struct_relationship_comp_edge(?_,[]):- true.
```

The output in Dot syntax follows:

```
…
/* composite relationship */
    deptrole[shape=ellipse,width=.75,height=.5,label=drole];
    dept->deptrole[arrowhead=none];
…
```

9.2 Sample Output

To generate the diagram in Dot one needs to invoke the procedure **%VIZ_SCHEMA** over some schema, e.g. 'afterScott', and direct its output into GraphViz for dot to render. The result of **%VIZ_SCHEMA** is shown in Appendix "GraphVIZ / Dot Specification (afterScott)". This is rendered by Dot and its graphical output is in figure 9.2.

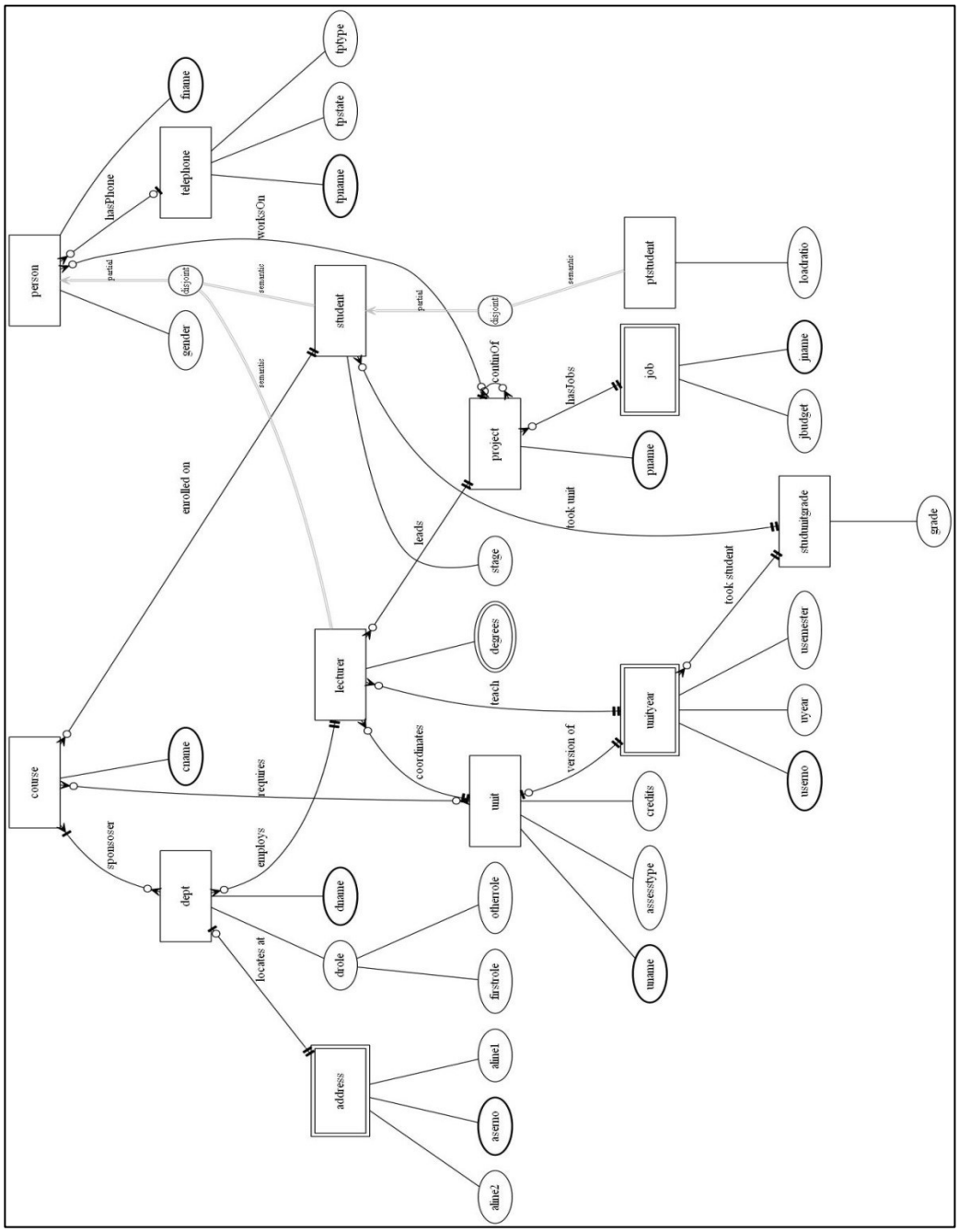


Figure 9.2 – Rendering of schema instance

9.3 Completeness and Correctness

The described framework encodes an EERM diagram through a number of assertions that describe the latter's structures and relationships. Furthermore a number of methods implement integrity constraints and other checks that support both the EERM encoding and the framework itself. Some of these have been presented earlier. The method **`%VIZ_SCHEMA`** of object **`SCHEMA`** generates a graph of the encoding in Dot syntax. This section will sketch a calculated consideration of this graphing process.

The consideration needs to address two issues: namely correctness and completeness. Of course an indicative measure of the mapping quality is running the GraphViz tool on the generated specification, to see if it is accepted. Specifically if a generated specification does not render then there is a problem. On the other hand, one cannot conclude from a valid drawing that the mapping is complete and correct.

9.3.1 The ISA relationship

For completeness of the *ISA* relationship mapping we need to show two things: first there are no F-logic *ISA* assertions (i.e. **::**) in the framework and the scope of EERM encoding that are not in the diagram generated, and the second is that there is no *ISA* relationship in the diagram that is not encoded in the framework.

If one fires the query **?- ?C:CLASS, ?CD:CLASS, ?C::?CD** on our example schema then one gets the instantiations for *ISA* assertions related to **CLASS** instances. Clearly the second row is correct but not needed for drawing.

```
?C = lecturer      ?Cd = person
?C = ptstudent     ?Cd = person
?C = ptstudent     ?Cd = student
?C = student       ?Cd = person
```

The following query and its result show how it weeds out any inferences by transitivity.

```
?- ?C:class, ?Cd:class, ?Cd::?C, \+((?_Ci:class, ?Cd::?_Ci, ?_Ci::?C)).
?C = person    ?Cd = lecturer
?C = student   ?Cd = ptstudent
?C = person    ?Cd = student
```

On the other hand the following query and its result show instances of the constraint **ISAPROPERTY**, clearly giving the same response as the previous query but having a different data source. (The same check could be expressed with the earlier predicate defined: i.e. **?C:CLASS[DIRECTASC->?C1]**).

```
?- ?_Isa:isaproperty[parentclass->?C, isaclass->?Cd].
?C = person    ?Cd = lecturer
?C = person    ?Cd = student
?C = student   ?Cd = ptstudent
```

Consequently we need to have the following two queries not instantiating.

```
?- ?_Isa:isaproperty[ parentclass ->? C, isaclass -> ?Cd],
   not((?C:class,?Cd:class,?Cd::?C)).
No.
?- ?C:class, ?Cd:class, ?Cd::?C,
   \+((?_Ci:class, ?Cd::?_Ci, ?_Ci::?C)),
   not((?_Isa:isaproperty[ parentclass -> ?C,isaclass -> ?Cd])).
No.
```

In procedure **%VIZ_ISA_ENTRY** the list of *ISA* instances is generated from the **ISAPROPERTY** and therefore the last two queries can safeguard the completeness of the *ISA* mapping.

There are a number of issues concerning the correctness of the *ISA* mapping. Firstly the **ISAPROPERTY** instances' properties (e.g. **PARENTCLASS** and **DISJ_OVER_FLAG**) need to have proper values assigned to them. This was explained in a previous section called "*ISA* template" (section 8.2.1.3). Secondly the drawing should be correct and legible. Going through the code it is, if not tedious, simple to go through the combinations and explain how each instantiation of arguments values produces the relative and unambiguous drawing fragment. Thirdly the relative nodes and edges are properly connected.

9.3.2 Classes, Structures & Attributes

The list of classes and structures are available as views in the framework: as **ALLCLASSES.CLASSLIST** and **ALLSTRUCTURES.STRUCTLIST** respectively.

The completeness and correctness of class instance drawing is relatively straightforward. The procedure **%VIZ_CLASS** uses the **CLASS** list as an argument when it calls **%VIZ_CLASS_ENTRY** and therefore it is complete. In terms of correctness the recursive procedure **%VIZ_CLASS_ENTRY** just prints the head of the **CLASS** list as a named box (i.e. a node) on each call until the list is exhausted.

The completeness and correctness of structure instances drawing is straightforward but comes in two batches. The procedure **%VIZ_STRUCTURE** does use the **STRUCTURE** list as an argument when it calls recursive procedure **%VIZ_STRUCTURE_ENTRY** but the latter only prints a double perimeter box if the argument's head is involved in a 'weak' relationship instance. The **STRUCTURE** instances that are not 'weak' have to be involved in a relationship instance-of type 'structure' and these are taken care of in procedure **%VIZ_STRUCT_RELATIONSHIP_COMP** and printed as an ellipse and connected to its entity in procedure **%VIZ_STRUCT_RELATIONSHIP_COMP_EDGE**. To check that all structures are involved in either 'weak' or 'structure' relationship instances the following denial query works.

```
?- ?S:structure,
   \+((?C:class,
       ?_Rc:refconstraint[(reltype->'structure'; reltype->'weak'),
       comp(?C,?_,?_,?_,?_,?_)->?S])).
No.
```

The query to check that all 'weak' and 'structure' relationship instances use structure instances follows:

```
?- ?C:class,
   ?_Rc:refconstraint[(reltype->'structure'; reltype->'weak'),
                      comp(?C,?_,?_,?_,?_,?_)->?S],
   \+((?S:structure)).
No.
```

The correctness of drawing the structures involved in 'weak' relationship instances is clear as the procedure checks the relationship type and if needs be prints the node details. (The weak relationship instance edge comes later). The correctness of drawing the composite artefact in a 'structure' relationship instance involves introducing the node as an ellipse and also an edge from the entity to this composite object (i.e. encoded as a structure).

The mapping of attributes is also straightforward and it is best to enumerate the two sources of attributes: these are **CLASS** and **STRUCTURE** instances. The mapping uses views, described earlier, to enumerate the attributes to draw – for example those that are declared in a class rather than inherited. These include the **CLASS** and **STRUCTURE** lists, and instances of object **GVIZ**. The procedure that caters for attribute drawing is called **%VIZ_ATTRIBUTE** and in turn calls procedure to draw single and multi-value attributes. For completeness one invokes a denial query that checks all **CLASS** and **STRUCTURE** instances are present (e.g. as **GVIZ** instances). Correctness has to deal with the depiction of each attribute and verify that the code does partition according to the signature declarations of **CLASS** and **STRUCTURE** instances. Correctness therefore also depends on the **CLASS** and **STRUCTURE** data signature too. Once a **CLASS** instance, or a **STRUCTURE** instance, is associated with a set of attributes, the latter are created as nodes whose properties also determine their rendition (e.g. oval, double oval, bold perimeter for primary key participation), and also an edge is built from the specific attribute to its **CLASS** instance.

9.3.3 Binary Relationships

This section considers the completeness and correctness of drawing relationship instances of type 'normal' and 'weak'. We recall that the framework converts, or expects the import, an *n*-ary relationship instance (with $n >= 3$) into n binary relationships. This transformation's validity was described in section 8.2.1.7 of chapter 8. Consequently only binary relationships are examined.

All relationship instances are instantiated by object **REFCONSTRAINT** and therefore for a binary relationship it has to have two components and each is arbitrarily assigned a '1' or a '2' – i.e. a serial number. Also, for each of its components it must hold that if a component starts at a class instance, for example, and leads to a second class instance, then the second component must have these instances inverted. Furthermore the lower and upper limits in each component must have one of of six possible values, without loss of generality. First note that object **UDO** is a direct ancestor of objects **CLASS** and **STRUCTURE**. The first denial queries below checks the serial number compliance of the relative relationship instance and that the relationships components go forward and back in terms of **CLASS** and **STRUCTURE** instances. The second denial query checks that the lower and upper limits of each relationship instance component take a proper combination of values (i.e. supported by procedure **%VIZ_REL_EDGE_LU** that has been described earlier).

```
?- ?I::udo.
?I = class
?I = structure
Yes.
?- ?R:refconstraint[reltype->'normal'; reltype->'weak'],
   \+(( ?R[comp(?C1,?_,?_,?_,?_,1)->?C2,
           comp(?C2,?_,?_,?_,?_,2)->?C1],
        ?C1:udo, ?C2:udo )).
No.
?- ?R:refconstraint[reltype->'normal'; reltype->'weak'],
   \+(( ?R[comp(?_,?_,?_,?L,?U,?_)->?_],
        %viz_rel_edge_lu(?L,?U,?_) )).
No.
```

The checking by the first denial ensures that each relationship instance has a component with serial number '1' assigned and completeness is ensured by enumerating through procedures **%VIZ_REL_NORMAL** and **%VIZ_REL_WEAK**. Since the restriction is made on serial number being assigned '1' then it is obvious that a relationship instance is only invoked once and therefore only one edge is drawn for it.

The correctness of the edge drawing is given by the fact that a relationship instance component's lower and upper limits are what are being expected and an unambiguous drawing is generated for each. Without loss of generality only one component contributes for the caption on an edge. Also the nodes, i.e. entities of weak entities, are present and already have had their properties defined in an earlier part.

Although 'normal' and 'weak' relationships instances are catered by different procedures they actually only differ in their data collection part and therefore share common procedures.

9.4 EERM to ODMG ODL mapping

In structured methodologies for system design the step between logical to physical data design has to have high adequacy cover, if not complete, and be correct. Another aspect is the reversibility of the translation. Unfortunately in practice this conversion is mostly unidirectional; i.e. from diagram to schema.

In this research conceptual data design is represented with an EERM model, and it has been indicated earlier that our EERM model is built and exported from a CASE tool. As for the logical and physical data design it is through an object-oriented data model that specifically uses ODMG ODL constructs.

This section explains an automated procedure that converts an EERM model encoded, verified, and validated in our framework into a schema expressed with ODL constructs. These are then executed by an OODBMS and an object database is created with a schema instantiated from a conversion output.

The reasons for automating this conversion are as follows. Firstly, the process is quick especially when compared to hand coding. Secondly, the conversion is systematic and disciplined, for example, in terms of the design patterns it creates, and in naming schema artefacts. Thirdly, the conversion is a basis on which the structural conversion is augmented with transitional artefacts (e.g. triggers and procedures). Fourthly, the conversion can provide comments and indicators that the data designer reads and then addresses. Another advantage for automated conversion is that it is relatively straightforward to cater for minor idiosyncrasies in the ODL syntax due to different implementations.

The dialect of ODMG ODL chosen as a target for this conversion is EyeDB [VIERA99]. The EyeDB provides decent compliance with ODMG ODL and has also implemented ODMG OQL. EyeDB has support and programming interfaces with C++ and Java. Currently EyeDB is in an open-source development regime.

9.4.1 The Problem Definition

The aim of the conversion is to build a schema with object-oriented data model constructs that implements the EERM diagram by reading it, validating it and mapping it into our framework. The conversion has to have a high-level of completeness and correctness.

The main construct in the ODL is the class. In an ODL class one implements: the *ISA* relationships, attributes, binary relationships, constraints (for example unique and not null), and indices (physical constructs). Useful constructs in ODL for this conversion are the forward directive and enumerated type definition.

In our framework, which is a systematic repository for an EERM diagram, a number of checks, conversions, and constraints are enabled. Furthermore our framework builds a number of views (i.e. instantiating rules) that make the conversion easier to manage. Also, and as was the case when generating a diagram with GraphViz, one had to bear in mind two things during conversion: that inherited properties are excluded from a new class specification; referential constraints for binary relationships have to be implemented differently in each respective class.

Finally, if there is a construct in the EERM diagram that is not convertible into EyeDB ODL then the conversion is to make it clear whether to continue with an "approximation" or to halt altogether.

9.4.1.1 General Procedure

An object schema method, called **`%ODL_SCHEMA`**, generates the ODL constructs that are faithful to the encoded EERM. Prior to calling this method one has to confirm that all of the integrity constraints entrenched in the framework and EERM diagram encoding are adhered to for the mapping to start. Also, and if applicable, any n-ary relationship with $n > 2$ needs to be converted prior to start of conversion with the patterns illustrated in an earlier and any relationship attributes are home in an appropriate entity.

A high-level description of the method **`%ODL_SCHEMA`** follows. First the conversion starts with basic schema details dumped into the leading part of the output. The second part ensures that the user-defined enumerated types are specified, and all classes are 'forward' referenced (e.g. needed to address the reciprocal nature of any relationship construct). The final part creates the ODL's class constructs that implement the EERM's entities, weak entities, *ISA* relationships, relationship constraints, and other constraints. During the latter phase the conversion makes use of a number of instantiated rules that generate objects each of which is an instance-of object **`ODMG`**.

There are two techniques on how to apply EyeDB ODL constructs to build the object base. A rather straightforward one is to compose all constructs and then submit as a whole (e.g. batch mode). Another technique is to build classes incrementally in the following sequence classes, attributes, convert relative attributes to relationships, and introduce constraints one at a time. ODMG ODL standard does not cater for these types of constructs – i.e. in SQL these would be the **ALTER TABLE** and **ALTER CONSTRAINT** constructs. Although EyeDB does allow incremental build up in its ODL processing, but not by using the like of an **ALTER** command, it has occasional issues. Consequently the batch mode had to be used.

When the procedure runs successfully its output is directed into a text file. EyeDB's ODL utility, called **EYEDBODL**, is invoked and takes as input the database name to build the schema in, and the schema specification file just created by **%ODL_SCHEMA**. The EyeDB tool to create a database is called **EYEDBADMIN**. The following is a script, with most output stripped, on a Linux system that illustrates this. (Also Appendix "EyeDB ODL Specifications (afterScott)" and "EyeDB processing of afterScott schema Specifications" have a copy of the full ODL file specs and the output of the **EYEDBODL** utility run for the ODL file spec).

```
j@a1:~/.../eyedb/afterscott$ eyedbadmin database create afterscott
…
Done
j@a1:~/.../eyedb/afterscott$ eyedbodl -d afterscott -u ascottschema.odl
Updating ' ascottschema ' schema in database afterscott...
Adding class person
…
Done
```

The leading part of the conversion process is taken care of by a procedure called **%ODL_HEADER**. It just reads schema details from the framework title objects and prints them. Its code follows:

```
%odl_header :-
  write('// schema name & version: ')@_prolog,
  write(schema.schemaname)@_prolog,
  writeln (schema.schemaversion)@_prolog.
```

9.4.1.2 Domains and Forward references

ODMG ODL allows for enumerated types definition through the **ENUM** directive. Our framework allows for such artefacts and these are called domains and are instances of object **DOMAIN**. The following is an example of a domain instance in our framework:

```
sex:domain [ enum -> {'male','female'} ].
```

There are two restrictions to cater for: firstly our framework bestows a string data type of each domain instance enumeration; secondly EyeDB assumes that all strings are unique across all domain instances. In our framework there is a view with a list of all **DOMAIN** instances and is called **ALLDOMAINS.DOMAINLIST**. The following code explains its instantiation.

```
alldomains[domainlist->?L]  :- ?L = collectset{ ?D | ?D:domain }.
```

The outermost method calls **%ODL_ENUM** with **ALLDOMAINS.DOMAINLIST** as an argument. The latter procedure reclusively builds an **ENUM** directive for each **DOMAIN** instance in the argument list. The enumerated type's content is built by a **COLLECTSET** predicate (and unified with the variable **?EL**) and is individually printed by calling another recursive procedure **%ODL_ENUM_BODY** on these list of values (i.e. in **?EL**). This procedure needs to build a comma delimitated list of values that comprise the **ENUM** instance and this explains the three rules required to define **%ODL_ENUM_BODY** rather than the usual two (i.e. last entry does not require a comma).

```
%odl_enum([?Head|?List]) :-
  write('  enum ')@_prolog, write(?Head)@_prolog, writeln(' { ')@_prolog,
  ?El=collectset{ ?I |?I=?Head.enum },
  %odl_enum_body(?El),
  writeln('  }; ')@_prolog,
  %odl_enum(?List).
%odl_enum([]) :- true.

  %odl_enum_body([?Enum|[]]):- !,
    writeln(?Enum)@_prolog.
  %odl_enum_body([?Enum|?List]):-
    write(?Enum)@_prolog, writeln(',')@_prolog,
    %odl_enum_body(?List).
  %odl_enum_body([]) :- true.
```

The procedure generates the following ODL code for the **SEX DOMAIN** above.

```
enum sex {     female,      male    };
```

The ODMG standard specification allows for pre-definition of classes and interfaces; that is only their name is defined. This allows for a class specification to have references to another class that has not, at that point in time, been defined fully to the compiler. These are called 'forward declarations' in the ODL specifications. The outermost method calls the recursive **%ODL_FORWARD** procedure with a list of **CLASS** instances to build their forward declaration (see below). Actually the procedure is called twice: the first time will all **CLASSES** and the second time with all **STRUCTURES**. We recall that the latter list has both weak entities and structures used to build composite objects. Although ODMG ODL has its own 'structure'

construct to build composite objects EyeDB builds these through references to other classes so the code to generate is identical (more detail in the contiguous sections).

```
%odl_forward_ref([?Cl|?Rl]) :-
    write('class ')@_prolog,write(?Cl)@_prolog,writeln(';')@_prolog,
    %odl_forward_ref(?Rl).
%odl_forward_ref([]) :- true.
```

The following is part of the forward declarations generated.

```
class course;
class dept;
class lecturer;
```

9.4.1.3 ODL Class Head

The method **%ODL_SCHEMA** calls the recursive procedure **%ODL_PRINT_CLASSES** twice to print the ODL class constructs with the procedure's sole argument being a list of EERM artifacts. In the first case it is a list of entities that are found in **ALLCLASSES.CLASSLIST**, and in the second case it is the list of **ALLSTRUCTURES.STRUCTLIST** where sifting of the weak entities and composite structures is possible. The procedure is invoked on a list of entities and on each item of this list it calls procedure **%ODL_PRINT_CLASS_HEAD** with the entity instance as its sole argument.

In **%ODL_PRINT_CLASS_HEAD** a simple decision is taken on whether the current item being processed has a descendent or not (i.e. is path expression **?HEAD.DIRECTASC[]** defined?). If it is then the single and direct ancestor is included with ODL's **EXTENDS** directive. In either case the procedure calls **%ODL_PRINT_CLASS_BODY** with the current item as the sole argument to print the class properties (e.g. attributes and relationships) within the mandatory brace tokens. The following is the procedure's declarative code:

```
%odl_print_class_head(?Head):-
   if   ( ?Head.directasc[] )
   then ( write('class ')@_prolog,  write(?Head)@_prolog,
          write(' extends ')@_prolog, write(?Head.directasc)@_prolog,
          writeln(' { ')@_prolog,
          %odl_print_class_body(?Head),
          writeln(' } ;')@_prolog  )
   else ( write('class ')@_prolog, write(?Head)@_prolog,
          writeln(' { ')@_prolog,
          %odl_print_class_body(?Head),
          writeln(' } ;')@_prolog ).
```

It is to be noted that ODMG ODL does not offer distinctive artefacts for entities and weak entities. Consequently the ODL **CLASS** artefact is used for both. ODMG ODL does provide the **STRUCTURE** artefact and it is useful to build composite objects; but EyeDB ODL does not support this syntax and instead uses the **CLASS** artefacts for these too (next subsection

explains exactly how). As regards inheritance ODMG ODL only supports single inheritance between its **CLASSES**. Furthermore the *ISA* constraint implementation in ODMG ODL is limited to disjoint, semantic and partial. Therefore all other combinations are not supported directly; i.e. triggers can address some variants but these are not found in the ODMG ODL standard.

Examples of the code generated by **%ODL_PRINT_CLASS_HEAD** follows (but without any output from **%ODL_PRINT_CLASS_BODY**):

```
class job { … };
class lecturer extends person { … };
```

9.4.1.4 ODL Class body – Attributes

The procedure **%ODL_PRINT_CLASS_BODY** takes an EERM entity as an argument one at a time (i.e. and it is the same one as that of the calling procedure - **?HEAD**). The procedure then systematically sifts for its simple & single attributes, simple & set attributes, structure & single attributes, structure & set attributes, 'one' side of a binary relationships, 'many' side of a binary relationships, primary key constraints, and 'not null' constraints. The procedure coding follows but it is to be noted that the second argument of each call passes a keyword for each type of class property to generate (e.g. '**SETATTR**' for simple & set). The procedures for finding and directing the printing of properties of a class are kept as straightforward as possible. In fact in each of these procedures there is a heavy dependence on a view specifically built for it. In this section only the first four procedures are of interest.

```
%odl_print_class_body(?Head):-
      %odl_print_class_simple_single_attr(?Head,'attr'),
      %odl_print_class_simple_set_attr(?Head,'setattr'),
      %odl_print_class_structure_single_attr(?Head,'structattr'),
      %odl_print_class_structure_set_attr(?Head,'setstructattr'),
      %odl_print_class_oneside_relationship(?Head,'onesiderel'),
      %odl_print_class_manyside_relationship(?Head,'manysiderel'),
      %odl_print_class_pk(?Head),
      %odl_print_class_notnull_constraint(?Head,'notnull').
```

The procedure **%ODL_PRINT_CLASS_SIMPLE_SINGLE_ATTR** takes two arguments. The first is the EERM entity and the second the type of property being sifted; i.e. **?PT** unifies with '**ATTR**'. The procedure then builds a list of properties of the entity, passed as the first argument, that are simple & single through the **COLLECTSET** aggregate predicate. This list and the two arguments of the calling procedures are passed on to a recursive procedure named **%ODL_PCSA_ATTRLIST** and for each item on the simple & single list it prints ODMG ODL

attribute directive. (Please recall that **FWRITE** predicate is shorthand for a list of **WRITE** predicates). Before the data type of an attribute is printed a procedure, named **%DT_MAP**, is called to map the entity's attribute data type and cardinality annotations into the closest data type in ODMG (and EyeDB) ODL. Further below there is an example attribute – ODL class **DEPARTMENT** and attribute **DNAME**.

```
%odl_print_class_simple_single_attr(?Head,?Pt) :-
     ?Cssa = collectset{ ?A | ?Head(?A,?Pt):odmg },
     %odl_pcsa_attrlist(?Head,?Pt,?Cssa).
%odl_print_class_simple_single_attr(?_,?_):- true.

  %odl_pcsa_attrlist(?Head,?Pt,[?Ha|?Rlist]) :-
     ?Head(?Ha,?Pt):odmg[attrtype->?Dt,attrcard->?Ac],
     %dt_map(?Dt,?Ac,?Dtm),
     fwrite('attribute ', ?Dtm, ?Ha, ';'),
     %odl_pcsa_attrlist(?Head,?Pt,?Rlist).
   %odl_pcsa_attrlist(?_,?_,[]) :- true.

%dt_map(integer, single, int)         :- !,true.
%dt_map(integer, set,    int)         :- !,true.
%dt_map(float,   single, double)      :- !,true.
%dt_map(float,   set,    double)      :- !,true.
%dt_map(string,  single, string)      :- !,true.
%dt_map(string,  set,    'char[50]') :- !,true.
%dt_map(?Dt1,    ?_Card, ?Dt2)        :- ?Dt1=?Dt2,!,true.
```

In the case of attributes that are simple & set the procedures, whose head is **%ODL_PRINT_CLASS_SIMPLE_SET_ATTR**, are very similar to those of simple & single. The views have instances whose identifier contains '**SETATTR'**. The main difference is that the output pattern, to implement the set part, in ODL, requires "**SET < DATA TYPE > ATTRIBUTE_NAME;**". This is aided by the data type mapping procedure **%DT_MAP** explained earlier. Further below there is an example attribute – ODL class **LECTURER** and attribute **DEGREES**.

It is important to specify the views that support these procedures. For each class attribute its data type signature is used to extract required details. For '**ATTR'** and '**SETATTR'** type of **ODMG** instances (rules definition follow) the query of each view is similar and their distinction is based on the upper limit of an attribute; i.e. if it is '1' then it forms part of the '**ATTR'** view otherwise it is in the '**SETATTR'** view. In either view two further tests are made: first the data type of the attribute must be a basic data type or an instance-of **DOMAIN** (e.g. enumerated type); second that the attribute is not inherited from an ancestral entity. When the procedure is invoked on instances of **ALLSTRUCTURE.STRUCTLIST** similar views exists for **STRUCTURES**; i.e. these are defined with instances of **STRUCTURE** rather than **CLASS**.

```
?C(?A,'attr'):odmg[attrtype->?Dt,attrcard->'single'] :-  // single attribute
    ?C:class[?A{?Low:?Up}*=>?Dt],
    ?Up=1,(?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
    if ( ?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt] ) then ( false ) else ( true
).

?C(?A,'setattr'):odmg[attrtype->?Dt, attrcard->'set'] :- // set attribute
  ?C:class[?A{?Low:?Up}*=>?Dt],
  ?Up!=1, (?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
  if ( ?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt] ) then ( false ) else ( true ).

?C(?A,'structattr'):odmg[ attrtype->?Dt, attrcard->'single']:- // single str
  ?C:class[?A{?Low:?Up}*=>?Dt],
  ?Up=1, not(?Dt=?C;?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
  if (?_:refconstraint[reltype->'structure', comp(?C,?A,?_,?_,?_,?_) -> ?S],
      ?S:structure)
  then ( if (?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt])
         then ( false )
         else ( true  ) )
  else ( false ).

?C(?A,'setstructattr'):odmg[ attrtype->?Dt, attrcard->'set' ] :- // set str
  ?C:class[?A{?Low:?Up}*=>?Dt],
  ?Up!=1, not(?Dt=?C;?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
  if (?_:refconstraint[reltype->'structure', comp(?C,?A,?_,?_,?_,?_) -> ?S],
      ?S:structure)
  then ( if (?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt])
         then ( false )
         else ( true  ) )
  else ( false ).
```

The views for **STRUCTATTR** and **SETSTRUCTATTR** are slightly more involved (rules definition are found above). Apart from checking that an attribute's return data type is not a basic type nor an instance-of **DOMAIN** the rules instantiation demands that there is a **REFCONSTRAINT** instance, of type structure as in **?_:REFCONSTRAINT [RELTYPE -> 'STRUCTURE']**, that relates the composition between a **CLASS** attribute and a **STRUCTURE** (taking the role of a composite object part). If the attribute appertains to a **STRUCTURE** then views are slightly different as the return type of a **REFCONSTRAINT** composition is either a **CLASS** or a **STRUCTURE** instance (i.e. **?_:REFCONSTRAINT [COMP(?S,?A,?_,?_,?_,1) -> ?SR], (?SR:STRUCTURE;?SR:CLASS)**).

To print the entity's structure attributes whether set or singleton, the procedure collects these for every **CLASS** and **STRUCTURE** and then recursively processes these to print the respective ODL attribute directive. The code for set & structure is given next:

```
  %odl_print_class_structure_set_attr(?Head,?Pt) :-
      ?Cssa = collectset{ ?A | ?Head(?A,?Pt):odmg },
      %odl_pcssa3_attrlist(?Head,?Pt,?Cssa).
      %odl_print_class_structure_set_attr(?_,?_):- true.

      %odl_pcssa3_attrlist(?Head,?Pt,[?Ha|?Rlist]) :-
          ?Head(?Ha,?Pt):odmg[attrtype->?Dt, attrcard->?Ac],
          fwrite(' attribute ', ?Ac,' < ',?Dt, '* > ', ?Ha, '; '),
          %odl_pcssa3_attrlist(?Head,?Pt,?Rlist).
      %odl_pcssa2_attrlist(?_,?_,[]) :- true.
```

An example of the code generated for these cases is found below in class **DEPARTMENT** and attribute **DROLE**. As remarked earlier EyeDB does not support structures of ODMG ODL but implements them with classes. In EyeDB we convert an ODMG "structure" into a class, say class **CS**, and when this **CS** is required for an attribute's type, called **ACS** in class **CM**, then we specify the attribute as a literal set of **CS** object identifiers. This implies that although instances of **CS** have identity, the set of **CS** in attribute **ACS** of class **CM** is inline (i.e. stored as a literal value).

```
class dept {
  attribute string dname;
  attribute set < deptrole * > drole;
class lecturer extends person {
  attribute set <degree> degrees; … };
```

9.4.1.5 ODL Class body – Relationships

To generate a binary relationship construct in ODMG requires two definitions: one on each participating **CLASS** instance. Also each participation of these instances has only two options in terms of cardinality: that is either 'one' or 'many'. Along these lines, two views are defined to support a class participation in binary relationships. To determine whether participation is of type 'one' or 'many' one checks an attribute's upper limit in its data type signature. For an attribute to be included in a relationship view it needs to be covered by an instance-of **REFCONSTRAINT** whose type is either 'normal' or 'weak'.

```
?C(?A,'onesiderel'):odmg
[relcard->'one',reltoclass->?Dt,reltoattr->?A2,relref(?Line)->?Rc,
 reltype->?Rt]
:- ?C:class[?A{?_:?Up}*=>?Dt], ?Up=1,
   ?Rc:refconstraint
   [reltype->?Rt, (?Rt='normal';?Rt='weak'),
    comp(?C,?A,?_,?_,?_,?Line)->?Dt,
    comp(?Dt,?A2,?_,?_,?_,?_)->?C],
    ?A!=?A2.

?C(?A,'manysiderel'):odmg
[relcard->'many',reltoclass->?Dt,reltoattr->?A2,relref(?Line)->?Rc,
 reltype->?Rt]
:- ?C:class[?A{?_:?Up}*=>?Dt], ?Up!=1,
   ?Rc:refconstraint
   [reltype->?Rt, (?Rt='normal';?Rt='weak'),
    comp(?C,?A,?_,?_,?_,?Line)->?Dt,
    comp(?Dt,?A2,?_,?_,?_,?_)->?C],
    ?A!=?A2.
```

The two heading procedures are applied to every **CLASS** and **STRUCTURE** instance. For example, when a **CLASS** instance (i.e. unified to **?HEAD**) and view type (i.e. '**MANYSIDEREL**' is unified to **?PT**) are passed to procedure **%ODL_PRINT_CLASS_MANYSIDE_RELATIONSHIP** it creates a list of **ODMG** instances of type '**MANYSIDEREL**' through the respective view. The same

procedure then calls a recursive procedure, called **%ODL_PCOMR_RELLIST**, with three arguments: namely a **CLASS** instance, the component cardinality unified to **?PT**, and the list of relationships instances associated with this class and cardinality. In **%ODL_PCOMR_RELLIST** details from the **ODMG** view instance are extracted and consequently the syntactic structure of the **RELATIONSHIP** and its **INVERSE** construct are printed.

```
%odl_print_class_oneside_relationship(?Head,?Pt) :-
    ?Cssa = collectset{ ?A | ?Head(?A,?Pt):odmg },
    %odl_pcosr_rellist(?Head,?Pt,?Cssa).
%odl_print_class_oneside_relationship(?_,?_) :- true.

  %odl_pcosr_rellist(?Head,?Pt,[?Ha|?Rlist]) :-
     ?Head(?Ha,?Pt):odmg
     [reltoclass->?C2, reltoattr->?A2, relref(?Line)->?Rc],
     fwrite('relationship ', ?C2, ' * ', ?Ha)
     fwrite('inverse ', ?C2, '::', ?A2, '; '),
     %odl_pcosr_rellist(?Head,?Pt,?Rlist).
  %odl_pcosr_rellist(?_,?_,[]) :- true.


%odl_print_class_manyside_relationship(?Head,?Pt):-
    ?Cssa = collectset{ ?A | ?Head(?A,?Pt):odmg },
    %odl_pcomr_rellist(?Head,?Pt,?Cssa).
%odl_print_class_manyside_relationship(?_,?_):- true.

  %odl_pcomr_rellist(?Head,?Pt,[?Ha|?Rlist]) :-
     ?Head(?Ha,?Pt):odmg
     [reltoclass->?C2, reltoattr->?A2, relref(?Line)->?Rc],
     fwrite('relationship set < ', ?C2,' * > ', ?Ha),
     fwrite('inverse ', ?C2, '::', ?A2, '; '),
     %odl_pcomr_rellist(?Head,?Pt,?Rlist).
  %odl_pcomr_rellist(?_,?_,[]) :- true.
```

Examples of the code generated by procedures to generate binary relationship constructs follow:

```
class dept { …
  relationship address * mainoffice inverse address::adept;
  relationship set < course * > sponsors  inverse course::sponsorer; };
```

9.4.1.6 ODL Class body – Constraints

The framework supports a number of constraints and in this section we address two that are directly supported by ODMG and EyeDB ODL namely the primary key set and the 'not null'.

To print the primary key set of a class a procedure is invoked, called **%ODL_PRINT_CLASS_PK**, for every class and structure in the framework. The invocation then builds a list of attributes that compose the primary key set of the class being processed and calls the recursive procedure **%ODL_PRINT_CLASS** to print this list in a comma delimited format as required by the ODMG ODL syntax. Unfortunately EyeDB only accepts primary key set with a single attribute. Also the constraint, implemented as **UNIQUE** in EyeDB, requires a corresponding physical index definition to be effective. The following procedure is tailored for EyeDB reality

but requires minimal changes to properly define primary key constraints with a set of attributes.

```
%odl_print_class_pk(?C) :-
  ?Pkl=collectset{?A|?Pk:pkconstraint,?Pk.classname=?C,?Pk.attrlist=?A },
  %odl_print_class_pk_line(?Pkl).

  %odl_print_class_pk_line([?Pk|?_L]) :-
        write('  constraint<unique> on ')@_prolog,
        write(?Pk)@_prolog,write('; ')@_prolog,
        write('index on ')@_prolog, write(?Pk)@_prolog, writeln(';')@_p.
  %odl_print_class_pk_line([]) :- true.
```

Examples of the code generated by **%ODL_PRINT_CLASS_PK** follows:

```
class course { …
  constraint <unique> on cname; index on cname;
  constraint <notnull> on sponsorer; };
```

The framework supports candidate key constraints too. The conversion is almost identical to the procedure used for primary key except the collection aggregate looks for instances of **CKCONSTRAINT**. In EyeDB the candidate key has to be defined with the same construct as that of the primary key set.

In the example above there is an instance-of the 'not null' constraint. To generate the respective 'not null' constraints in each ODMG class we need to populate two views. The rule checks each attribute signature for a lower cardinality limit set to '1' and it ensures that it is not an inherited attribute. This applies to all attributes – even those taking part in referential constraints. The rule for **CLASSES** follows:

```
?C(?A,'notnull'):odmg[ consname -> 'notnull' ]  // not null constraints
:- ?C:class[?A{?Low:?Up}*=>?Dt], ?Low=1,
   if ( ?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt] ) then ( false ) else ( true
).
```

To print the 'not null' constraint the procedure **%ODL_PRINT_CLASS_NOTNULL_CONSTRAINT** is called and it creates a list of attributes from the view just defined that require the 'not null' constraint in a class instance passed as an argument. The procedure then calls a recursive procedure, called **%ODL_PCNNC_CONSLIST**, to print the ODMG ODL **NOTNULL** constraint.

```
    %odl_print_class_notnull_constraint(?Head, ?Pt):-
        ?Cssa = collectset{ ?A | ?Head(?A,?Pt):odmg },
        %odl_pcnnc_conslist(?Head,?Pt,?Cssa).
     %odl_print_class_notnull_constraint(?_,?_):- true.

        %odl_pcnnc_conslist(?Head,?Pt,[?Ha|?Rlist]) :-
             ?Head(?Ha,?Pt):odmg,
             write('  constraint<notnull> on ')@_prolog,
             write(?Ha)@_prolog,  writeln('; ')@_prolog,
             %odl_pcnnc_conslist(?Head,?Pt,?Rlist).
        %odl_pcnnc_conslist(?_,?_,[]) :- true.
```

9.4.2 %odl_schema Output

Once the **%ODL_SCHEMA** method, of object **SCHEMA**, is invoked the EyeDB ODL specifications are generated. These specifications are based on what the framework has read in and encoded, and in turn applying transformations and checks carried on the EERM model. The specifications are written into a text file and passed, as a script, to EyeDB ODL program.

The actual output (i.e. specification) for our non-trivial schema test EERM is found in Appendix "EyeDB ODL Specifications (afterScott)". In Appendix "EyeDB processing of afterScott schema Specifications "one finds the verbose comments of **EYEDBODL** on processing this specification file.

9.4.3 Completeness and Correctness

The framework developed here encodes an EERM diagram through a number of facts and rules that describe the diagram's structures and relationships. Furthermore a number of procedures implement integrity constraints and other checks that support both the EERM encoding and the framework itself. Some of these have been presented earlier. The method **%ODL_SCHEMA** of object **SCHEMA** generates a script with EyeDB ODL constructs. In this section intends to sketch an evaluation of this conversion process.

The evaluation needs to address two issues: namely correctness and completeness. An indicative measure of the conversion quality is running the EyeDB parser on the ODL syntax generated. Specifically if syntax does not pass then surely there is a problem. On the other hand if it compiles it is nonetheless insufficient to state the conversion is either complete or correct.

9.4.3.1 Converting the domains, entities, and weak entities

The completeness and correctness of the list of all domain instances defined in the framework is self-evident from the query used to build it. Nonetheless there are two correctness issues that need discussion other than the correctness of the procedures. First is the fact that each **DOMAIN** instance has a set of values and these values have to be of string data type. Therefore data typing checks, discussed in the next section, have to validate this. Second and this is more a requirement of the target environment, i.e. EyeDB ODL, requires that an element of a **DOMAIN** instance is distinct over all domain instances. This check requires the following

denial query, i.e. there are no two elements that are the same if domain instances are different, to fail:

```
?- ?D1:domain, ?D1.enum=?I1, ?D2:domain, ?D2.enum=?I2, (?D1!=?D2,?I1=?I2).
```

The forward reference directive is a rather simple procedure that converts two lists of artefacts (i.e. **CLASS** and **STRUCTURE** instances) into an ODL class definition but devoid of any details (e.g. attributes, and constraints). The procedure is recursive and traverses the list passed to print a keyword (i.e. **CLASS**) and the current head of the list. It is important to make a point about treating the **CLASS** and **STRUCTURE** list with the same ODL artefact – i.e. class. Inevitably once the EERM's entity and weak entities are defined with the same construct their distinction is lost on conversion. That is one cannot state that a **CLASS** is an entity or a weak entity from its ODL definition. (This is discussed in a following section on reverse engineering an ODL schema).

To generate a complete **CLASS** definition that comprises *ISA* relationships, attributes, binary relationships, and constraints, the procedure starts in a similar way to the forward declaration just discussed. We know that for each item on the **CLASS** and **STRUCTURE** list an ODL class has to be generated. To do this each list is passed as an argument to a simple recursive procedure, **%ODL_PRINT_CLASSES**, which for each invocation invokes another procedure to systematically add the ODL class properties on the head item of its argument list and relating to relevant ODMG objects. Clearly the union of these two lists covers all of our EERM artefacts. Also, given our acceptance of choice, the conversion is correct.

9.4.3.2 Converting the ISA relationship

The first argument is on the modality of inheritance in our encoded framework model of an EERM diagram and in ODMG ODL **CLASS** construct. Since both implement single inheritance there is no issue and therefore ODMG ODL correctly specifies single inheritance. What is an issue is the fact that ODMG inheritance mechanism is of type disjoint, partial and semantic. If an *ISA* constraint is specified in the EERM model that is not encoded so then coercion into a mode acceptable by ODMG ODL is not tenable. The following denial if successful implies non-completeness:

```
?- ?Isa:isaproperty, \+(( ?Isa[disj_over_flag -> 'disjoint', totl_part_flag
-> 'partial', sema_pred_flag -> 'semantic'])).
No.
```

What can be done to improve conversion and leave 'hand coding' as a last resort? Some aspects are addressed by improving the ODMG ISA mechanism and also introduce a wider range of constraints (e.g. **CHECK** constraint). Also, and as stated earlier, a trigger specification language is useful to implement the *ISA* specification details. At this point any *ISA* constraint that is not disjoint, partial and semantic is reported during conversion.

For completeness of the *ISA* relationship conversion we need to show two things: first there are no F-logic *ISA* assertions (i.e. **::**) in the framework and the scope of EERM encoding that are not in the **CLASS** generated, and the second is that there is no *ISA* relationship in the conversion but not encoded in the framework. Consequently we need to have the following two queries not instantiating.

```
?- ?_Isa:isaproperty[ parentclass ->? C, isaclass -> ?Cd],
   not((?C:class,?Cd:class,?Cd::?C)).
No.
?- ?C:class, ?Cd:class, ?Cd::?C,
   \+((?_Ci:class, ?Cd::?_Ci, ?_Ci::?C)),
   not((?_Isa:isaproperty[ parentclass -> ?C, isaclass -> ?Cd])).
No.
```

In procedure **%ODL_PRINT_CLASS_HEAD** each instance-of class or structure is checked if its direct ancestor is define so as to introduce the ODL ISA assertion (i.e. **?HEAD.DIRECTASC[]**). Therefore one needs to ascertain that all *ISA* assertions are indeed caught with this filter.

```
?- ?C:class, ?Cd:class, ?Cd::?C, \+((?_Ci:class, ?Cd::?_Ci, ?_Ci::?C)),
   \+((?Cd.directasc=?C)).
No.
```

9.4.3.3 Converting the attributes

Two groups of procedures to print a class attribute, one for singleton and the other for 'set of' are mostly identical to each other and they differ only in output and on which views they read from. The outer procedure take two arguments that limit the scope on which **ODMG** objects are of interest, and then builds the relevant list to recursively processed and convert into ODL attributes.

For the completeness property we need to investigate two data sources. First is the familiar **CLASS** and **STRUCTURE** list of instances and this has been purported to be correct. Second is the list of relative **ODMG** objects per **CLASS** instance. The rule that generates the **ODMG** objects for singleton and non-inherited attributes of a **CLASS** instance follows (the others having been listed earlier):

```
?C(?A,'attr'):odmg[attrtype->?Dt,attrcard->'single'] :-  // single attribute
    ?C:class[?A{?Low:?Up}*=>?Dt],
    ?Up=1,(?Dt='string';?Dt='integer';?Dt='float';?Dt:domain),
    if ( ?C::?Sc,?Sc:class[?A{?Low:?Up}*=>?Dt] ) then ( false ) else ( true
).
```

What remains to be seem is that there are no instances of **ODMG** that are not assumed by the above rule. The following query checks for this:

```
?- ?C(?A,'attr'):odmg, ?C:class[ ?A{?_L:?U}*=>?Dt],  \+((?U=1,
(?Dt='string'; ?Dt='integer'; ?Dt='float'; ?Dt:domain) )).
No.
```

A similar check is done for every **CLASS** and **STRUCTURE** instance and for qualifier '**ATTR'** and '**SETATTR'**.

The correctness of the generated code requires a last check, namely the mapping from the EERM annotated data type of an attribute to an equivalent data type in ODMG ODL. This mapping is taken care of by procedure **%DT_MAP**. If the EERM annotation is data-type checked for appropriate values then this part of the conversion is complete and correct too.

Another two sets of procedures take care of single & structure and set & structure attributes. These two groups are very similar to those just described but each has its own view and code generating bit. In particular, the data view definition checks for a **REFCONSTRAINT** instance of type structure. The completeness tests just described are applicable here to. As for correctness the distinctive part is the code generation and this requires explanation. Firstly the composite object building has to be done through the EyeDB ODL **CLASS** construct; implying that the EERM structure is converted to a **CLASS**. Secondly although the composite part is considered as a 'part-of' the holding object, and consequently the former is a literal (i.e. no need of an object identifier), object identifiers are still bestowed. This extra specification is an idiosyncratic part of EyeDB ODL.

Finally, a similar query to one presented earlier checks that there are no **ODMG** instances (of type '**STRUCTATTR'** and '**SETSTRUCTATTR'**) that are not covered by the respective rules firing.

9.4.3.4 Converting the relationships

This section considers the completeness and correctness of converting relationship instances of type 'normal' and 'weak'; instances of type 'structure' are taken care on structured attributes. We recall that the framework, or expects the import, to convert an *n*-ary relationship instance

(with $n >= 3$) into n binary relationships. Consequently only binary relationships are examined.

All relationship instances are instances of object **REFCONSTRAINT** and therefore for a binary relationship it has to have two components. To build a complete list of relationship instances we use the aggregate **COLLECTSET** predicate.

In each of a relation's instance components it must hold that if a component starts at a class instance then the second component must return to it. Furthermore, the lower and upper limits in each component must have one of six possible values, without loss of generality. The denial queries required here are identical to those used for drawing the relationship with GraphViz.

We need to invoke a check that verifies that it is that's not the case that an **ODMG** instance is not caught by the relative view; e.g. **?C(?A,'ONESIDEREL'):ODMG**. This check is similar to one presented in the previous section.

The correctness of the conversion is given by the fact that a relationship instance's component's lower and upper limits are what would be expected and an unambiguous code fragment is generated for each. Although 'normal' and 'weak' relationships instances are catered by different group of procedures they actually only differ in their data collection part and therefore share common procedures.

```
?C(?A,'onesiderel'):odmg
[relcard->'one',reltoclass->?Dt,reltoattr->?A2,relref(?Line)->?Rc,
 reltype->?Rt]
:- ?C:class[?A{?_:?Up}*=>?Dt], ?Up=1,
   ?Rc:refconstraint
   [reltype->?Rt, (?Rt='normal';?Rt='weak'),
    comp(?C,?A,?_,?_,?_,?Line)->?Dt,
    comp(?Dt,?A2,?_,?_,?_,?_)->?C],
    ?A!=?A2.
```

9.4.3.5 Converting the constraints

The framework supports a number of constraints (the following query lists these).

```
?- ?Cl=collectset{?C|?C::constraint}.

?Cl = [check, ckconstraint, fdconstraint, isaproperty, notnull, oneonly,
pkconstraint, refconstraint]
Yes.
```

EyeDB and ODMG ODL does not cover all of these, in particular the **CHECK**, **FDCONSTRAINT**, and **ONEONLY**. Consequently completeness is lost. In this section we consider the

completeness and correctness properties of two constraints, i.e. 'not null' and 'primary key' that are supported by EyeDB ODL but have not yet been discussed.

The 'not null' constraint is easy to check for completeness. The relative views hold attribute whose definition has lower cardinality limit set to '1'; this applies to attributes that include binary relationships. One can then check that indeed any attribute with lower cardinality set to '1' is implied by the respective view. The procedure to generate the 'not null' constraint is quite simple and the code it generates is trivial to check for correctness.

EERM construct	Framework	EyeDB ODL	EERM-ODL map issues
Entity	class	class	
Weak entity	structure	class	
Relationship (n=2) w/o attr	refconstraint (i.e.normal, weak) & classes	class relationship and inverse construct, and not null constraint	
Relationship (n=2) + attr	refconstraint (i.e.normal, weak) & class	class relationship and inverse construct, attribute, and not null constraint	With resolving class introduced for many to many
Relationship (n>2)	refconstraint is converted to class & n binary relationships		Not supported
Attribute – simple & single	class | structure	class attribute	
Attribute – simple & set	class | structure	class attribute	
Attribute – structure & simple	class | structure refconstraint (i.e. structure)	class attribute whose type is an identifier to class (that implements the structure)	
Attribute – structure & set	class | structure refconstraint (i.e. structure)	class attribute whose type is set of identifiers to class (that implements the structure)	
Possible notes	domain	enumerated types	
Primary key	pkconstraint	class unique constraint	Partial (single attribute)
n/a	ckconstraint	class unique constraint	Partial (single attribute)
n/a	fdconstraint	class unique constraints	Partial (simple and single attribute)
ISA relationship	ISA constraint	class with constraint	Only disjoint, semantic and partial are implemented.
	Check constraint		Not directly supported
	Triggers	trigger	Not supported by ODMG ODL
n/a	Basic data types	basic data type	

Table: 9.1 – Summary of EERM, framework, and EyeDB ODL objects

The primary key constraint is relatively easy to check for completeness and correctness. If a constraint exists for a current class instance then its participating attributes are collected

from the relative constraint construct and ODL unique is built by a recursive procedure. Unfortunately in EyeDB ODL the **UNIQUE** construct takes one and only one attribute. Consequently it is not complete. The correctness, limited to a singleton, of the construct is straightforward to verify as the syntax is simple – it uses the **UNIQUE** construct. Another requirement is that a physical artefact, i.e. an index, is built with every **UNIQUE** constraint.

Candidate key set constraints, where one can assert that a primary-key constraint is its specialization, has the same arguments as primary-key set. On top of that there is an issue in EyeDB's implementation regarding correctness, this relates to the fact that there is no distinction between primary and candidate key definitions in the class definition.

9.4.3.6 Conversion summary

In table 9.1 one finds a summary of the objects, constructs and conversion from one to its corresponding artefact. It is important, at this point, to repeat the importance of having a high level of completeness because this will attenuate any coding required to address missing design artefacts.

9.4.4 Reverse Engineer an EERM Diagram from an ODL Schema

What about converting an ODL schema into an EERM diagram? It is an important question, or better still a pressing requirement from data designers, as a number of activities greatly benefit from this reverse engineering. For example, in database integration, i.e. between as yet independent data sources, this reverse engineering aids in quickly building a prototype schema. Unfortunately there are two issues. First it is not given that if an EERM is mapped into a schema and then immediately reverse engineer it into an EERM the later EERM has anything to do with the original EERM. (We are not referring to layout problems as perfidious as these can be). This is mostly to do with assumptions taken in the conversion, and the fact that some target constructs are overloaded. The second problem is that the schema in an active database would have a large number of artefacts that aid software development and data management. For example, the introduction of lookup tables raises the data quality of some attribute's assigned values.

In this study it is important to ask what is 'lost in translation', and effectively evaluate another aspect of the conversion. The previous table (i.e. 9.1) does give immediate indications.

One genera of loss comes for relationships with an order greater than two where it is converted into *n* binary relationships. While converting relationship there is another technique which is lossy and occurs when relationship attributes are lumped with other class attributes. Frankly there is little to do with these patterns and in fact many types of data models suffer this.

Another general type of loss appertains to ODMG and EyeDB ODL. For example an EERM's entities and weak entities are generally converted into respective class constructs. In EyeDB matters are more pronounced as ODMG's ODL structures are implemented with classes too and this effectively fudges further the distinction between weak entity and composite object. Another example is the *ISA* relationship implementation in OMG ODL which accepts no variety found in EERM diagrams.

There are some other issues, reported earlier, with the use of EyeDB ODL unique construct to implement primary key, candidate key and uniqueness constraint. Consequently there is no clue in the EyeDB data dictionary as to a constraint's real provenance.

At this point we assert that to attenuate the conversion and translation lossyness requires ODMG ODL to acquire better and wider constructs (e.g. triggers with incremental coding). Also conceptual design constructs, read from an EERM diagram, are read and maintained within the object base data dictionary.

9.5 Summary

This chapter has shown how this framework can accommodate a wide variety of EERM designs created, for example, in a CASE tool. The EERM constructs are an input to the framework. The framework allows a transformation of valid EERM artefacts into constructs that DBMSs are capable to handle. For example an *n*-ary relationship is converted into a resolving class instance, *n* binary relationships, and a set of constraints (e.g. primary and candidate keys). Another important feature is that additional artefacts are added to supplement the EERM model when it is read in. Two examples are functional dependences and candidate keys. Some issues with EERM do hinder its fuller utility and two need addressing: the first is some constructs require better, wider and more acceptable notation and meaning (e.g. aggregation). Also transitional constraints, not really part of traditional ERM, are a must with more involved modelling.

At this point the framework is capable to aid in database design. For example an EERM encoded, or read from an ASCII file, is converted in an ODMG ODL schema. In this study we have tailored the output to an ODMG dialect offered by EyeDB; but it is easy to convert to other ODMG ODL syntax flavours. The conversion has a wide coverage and takes non-trivial examples as shown with the schema on which it was tested on. Furthermore the conversion is aided by an extensive range of pre-conversion tests. Nonetheless there are some issues due to EyeDB idiosyncrasies and limitations, but an acceptable solution for meeting most of them was presented. Two issues with ODMG are its almost lame relationship construct, and absence of triggers to implement transitional constraints. In the case of relationships shortfall the translation and framework offers a lot. In the case of triggers we believe substantial progress needs to be introduced to their definition, maintenance, and management before automating the process of their coding.

Functionality possible through our framework is to redraw an EERM that has been read. The graphing package is sent a specification file, that includes design artefacts, and in turn it produces an image. This feature is most useful when an original EERM design is worked on in the framework; as in improving it or when integrating two designs (i.e. during a data-integration exercise).

In both mappings, apart from rigorousness and detail that is not available in database design literature for object-oriented databases, we presented issues of correctness and completeness for each mapping. Also we have shown the inadequacy of reverse engineering an ODMG ODL schema back to the original EERM model.

This work has had it start in Vella's [VELLA97] then with an early version of ODMG ODL; the latest publication is in [VELLA09]. Similar work is too general as can be seen in standard works; as El Masri [ELMAS10] and Teorey [TEORE05]. Another advantage of this work is its integration with other features of the framework; query modelling is dependent on structures read from an EERM model.

Chapter 10

Type Checking in Object Database Framework

10 Type Checking in Object Database Framework

In chapter four we have asserted, through literature reviews, that data typing and data type checking help develop readable, reliable, and efficient software. The object-oriented paradigm is strongly associated with a number of sophisticated data-typing features, and most of its adopters expect to have these. We have also reported how a mix of typing modalities, specifications, and checking have been tested for the object-oriented paradigm.

In F-logic one can associate data-type signatures with objects and there exists a number of data typing axioms. More interestingly the type signatures, applicable to inheritable and non-inheritable methods, are part of the inference process; e.g. data-typing rules and assertions interact with other objects assertions like those of *ISA* and instance-of.

In Flora-2 the data-type signature specifications are accepted but no implicit type checking is available and some type-inference rules are not evaluated; furthermore some F-logic axioms are not entrenched in its semantics – an example being a method's attribute contra-variance and co-variance. As regards type checking Flora-2 nonetheless does have a strong potential; we have already seen how F-logic and Flora-2 higher-order syntax mixes data and structure in query expressions. In fact the framework adopted here builds run-time type-checking procedures to implement basic data checking and inference.

In this section we start by describing a number of views that aid the type-checking process. The first type checking has to do with the cardinality constraints of methods. The next section deals with data-type checking of methods of user defined classes (e.g. classes and structures) and their instances. The later sections discuss object polymorphism type checking and F-bounded polymorphism.

10.1 Problem Definition

What are the data typing and inference requirements of our framework? Firstly, we need to check that the data expressions of our object base are proper in terms of the data signatures provided. Furthermore, because of the nature of F-logic we expect some form of overloading and parametric polymorphism. Within this requirement we have to include a check to ensure that all object assertions, especially instances of user-defined objects, are covered by at least

one signature. Secondly, we need to accommodate recursive data types and also recursive data structures (e.g. like lists of objects). Thirdly, we need the framework to work out the intricacies of data specification and typing in the presence of recursive types and subtyping realities.

10.2 Views and Flora-2 Data-Type Signatures

In our framework the most common data signature used for **CLASS** and **STRUCTURE** instances is the type ***=>**. These are called inheritable methods. In the following script the instance **GRAND** has a method signature for **FNAME**. Instance **GO1**, related to **GRAND** with the instance-of relationship (i.e. **:**), inherits the signature **=>** and **FNAME** takes a string through the **->** assignment. Methods denoted by **=>** are called non-inheritable methods. (The same applies to object **PO1** but inheriting from the **CLASS** instance **PERSON**).

The class **PARENT**, related to **GRAND** with an *ISA* relationship (i.e. **::**), inherits all the data signatures of the latter. Consequently any instance-of **CLASS** and **STRUCTURE** inherits its data signature and remains denoted with ***=>**. Any **CLASS** method denoted by **=>** is not inherited by the *ISA* relationship but passed on to its instances – e.g. these are useful for class attributes described in earlier chapters.

```
grand : class[ fname{1:1} *=> string].

go1:grand [fname -> "g1name"^^_string].

parent::grand.
child::parent.

?- parent[ fname *=> string ].
Yes.

parent[ livesat{1:1}  *=> string ].

?- parent[ ?M *=> ?Rt ].
?M = fname    ?Rt = string
?M = livesat  ?Rt = string
Yes.

po1:parent[fname -> "p1name"^^_string, livesat -> "p1home"^^_string].

?- po1[ ?M => ?Dt ].
?M = fname    ?Dt = string
?M = livesat  ?Dt = string
Yes
```

Another feature of a Flora-2 data signature is the cardinality constraints of a method. For example, in the method **FNAME** above the cardinality constraints specify a lower and upper constraint of '1' (one and only one) – i.e. **{1:1}**. Both limits take any integer with the lower

having to be less than, or equal, to the upper. Cardinality constraints in Flora-2 are more flexible than F-logic as the latter only allows singleton or set cardinality.

In F-logic one can also specify methods with arguments together with their data signature. The following script gives a simple example where a **CLASS** instance called **CC** has a method, called **CMTH5**, that takes one argument of type **PARENT** and returns an instance-of type **PARENT**. The effects of the instance-of and *ISA* relationships are identical for methods without arguments (and as presented earlier). Methods without an argument type in their signature are called scalar methods and those with arguments, as is **CMTH5**, are called arity methods.

```
cc:class.
cc[ cmth5(parent) *=> parent ].

  cco1:cc[ cmth5(po1) -> co1 ].
  cco2:cc[ cmth5(go1) -> po1 ].
```

F-logic allows data signatures to be overloaded too. In the following example, method **OVERLOAD** has two distinct signatures specified. Methods with arguments can also be overloaded.

```
c1:class[ overload *=> integer, overload *=> string].

?- c1[overload*=>?Dt].
?Dt = integer
?Dt = string
Yes.
```

10.2.1 Views for Classifying Methods for Data-Type Checking

The first set of views enumerates the scalar methods; methods that come without an argument in their signature. These capture either the inheritable and non-inheritable methods for instances of **CLASSES** and **STRUCTURES** – collectively instances of **UDO**. The following two rules build **SCALARMETHOD** and **INHSCALARMETHOD** property of each user defined instance (i.e. instances of **CLASS** or **STRUCTURE**).

```
?C[scalarmethod->?M]      :- ?C:udo[?M =>?_ ],  not compound(?M)@_prolog.
?C[inhscalarmethod->?M]   :- ?C:udo[?M *=>?_ ], not compound(?M)@_prolog.
```

The predicate **COMPOUND** returns true if the term (denoted by **?M**) is a composite term; in this case it would imply that the method comes with an argument. The partitioning is controlled by a difference in the data signature for inheritable (***=>**) and non-inheritable (**=>**) methods. The following query lists the inheritable methods of **CLASS** instance **PARENT**.

```
?- ?C=parent,?C[inhscalarmethod->?L].
?C = parent   ?L = fname
?C = parent   ?L = livesat
Yes.
```

Each of these methods is complemented with another property wherein the methods are included in a list (named **SCALARMETHODS** and **INHSCALARMETHODS)**. For example:

```
?C[inhscalarmethods->?Mlst :-
    ?C:udo, ?Mlst=collectset{?M|?C[?M*=>?_], not compound(?M)@_prolog}.
?- ?C=parent,?C[inhscalarmethods->?L].
?C = parent  ?L = [fname, livesat]
Yes.
```

The second set of views enumerates arity methods for instances of **UDO**; methods that come with an argument in their signature. These capture either of the inheritable and non-inheritable methods. The following two rules build **ARITYMETHOD** and **INHARITYMETHOD** property of each user defined instance (i.e. **CLASS** or **STRUCTURE**).

```
?C[aritymethod->?Mth]     :- ?C:udo[?M=>?_],
compound(?M)@_prolog,?M=?Mth(?_).
?C[inharitymethod->?Mth] :- ?C:udo[?M*=>?_],
compound(?M)@_prolog,?M=?Mth(?_).
```

The predicate **COMPOUND** for method, i.e. **?M**, must be true. Also note that method name assigned in the view is stripped of its argument data type (i.e. **?MTH** rather than **?M**). The partitioning is controlled by the different data signature for inheritable (***=>**) and non-inheritable (**=>**) methods. The following query checks whether **CMTH5** is an arity and inheritable method of **CLASS** instance **CC**.

```
?- cc[ inharitymethod -> cmth5 ].
Yes.
```

Each method is complemented with another property wherein the methods are included in a list (named **ARITYMETHODS** and **INHARITYMETHODS)**.

For each instance-of **UDO** there are collective views that gather its inheritable and non-inheritable methods; these views are called **INHMETHODS** and **METHODS** and the property is set based.

```
?C[methods -> ?M]     :- ?C:udo[scalarmethod -> ?M];
                         ?C:udo[aritymethod -> ?M].
?C[inhmethods -> ?M] :- ?C:udo[inhscalarmethod -> ?M];
                         ?C:udo[inharitymethod -> ?M].
```

The third set of views enumerate the overloaded methods for instances of **UDO**; methods that come with more than one return data type signature or one argument data type. These capture either of the inheritable and non-inheritable methods. The following four rules build the respective properties of each user defined instance (i.e. **CLASS** or **STRUCTURE**). The rules identify methods (i.e. **?M**) through previously defined views that have for example a different return data type.

```
?C[nameoverloadedmethods->?M]    :-
    ?C:udo[methods->?M],
    ?C[?M=>?D1], ?C[?M=>?D2], ?D1!=?D2.
?C[nameoverloadedmethods->?M]    :-
    ?C:udo[aritymethod->?M],
    ?M=?Mth(?_Pdt),?C[?Mth(?Pdt1)*=>?_], ?C[?Mth(?Pdt2)*=>?_],?Pdt1!=?Pdt2.
?C[nonnameoverloadedmethods->?M] :-
    ?C:udo[methods->?M],
    not ?C[nameoverloadedmethods->?M].

?C[inhnameoverloadedmethods->?M] :-
    ?C:udo[inhmethods->?M], ?C[?M*=>?D1], ?C[?M*=>?D2], ?D1!=?D2.
?C[inhnameoverloadedmethods->?M] :-
    ?C:udo[inharitymethod->?M],
   ?M=?Mth(?_Pdt), ?C[?Mth(?Pdt1)*=>?_], ?C[?Mth(?Pdt2)*=>?_], ?Pdt1!=?Pdt2.
?C[inhnonnameoverloadedmethods->?M]:-
    ?C:udo[inhmethods->?M],
    not ?C[inhnameoverloadedmethods->?M].
```

The following query checks if method **OVERLOAD** of **CLASS** instance **C1** is listed as overloaded.

```
    ?- ?C=c1, ?C[inhnameoverloadedmethods->?M].
    ?C = c1     ?M = overload
    Yes.
```

The last set of views address a method's argument and return data type contra-variance and co-variance. This is all the more necessary as Flora-2 does not support contra-variance and variance for methods. The two rules create objects whose identifiers are **AS(…)** and some of its properties include **CLASS** and **METHOD.**

```
as(?C,?M,?Pdt2,?Rdt,"noninh")
  [class->?C, method->?M, pdt->?Pdt2, rdt->?Rdt, inhtype->"noninh"]
  :- ?C:udo[aritymethods->?Mlst], member(?M,?Mlst)@_prolog(basics),
     ?C[?M(?Pdt)=>?Rdt], (?Pdt2::?Pdt;?Pdt2=?Pdt).

as(?C,?M,?Pdt2,?Rdt,"inh")
  [class->?C, method->?M, pdt->?Pdt2, rdt->?Rdt, inhtype->"inh"]
  :- ?C:udo[inharitymethods->?Mlst], member(?M,?Mlst)@_prolog(basics),
     ?C[?M(?Pdt)*=>?Rdt], (?Pdt2::?Pdt;?Pdt2=?Pdt).
```

An object is created for each arity method of a user-defined object instance. Furthermore if the return data type, i.e. **?RDT**, is a descendent of a **CLASS** instance then an **AS(…)** object is created for each of its ancestors and the **?RDT** property replaced by the *ISA* inference. For example, if the return type, **?PDT2**, is **CLASS PARENT** then another **AS(…)** object is created for **CLASS GRAND**. In the case when the argument data type is an ancestor of other **CLASS** instances then an **AS(…)** object needs to be created for each of its descendants. For example if an argument data type is **CLASS** instance **PARENT** then another **AS(…)** object needs to be instantiated for argument data type **CHILD CLASS**. The following query shows this view's evaluation.

```
?- ?C=cc,?M=cmth5,
   as(?C,?M,?Pdt2,?Rdt,"inh")
      [class->?C, method->?M, pdt->?Pdt2, rdt->?Rdt, inhtype->"inh"].
?C = cc    ?M = cmth5    ?Pdt2 = child    ?Rdt = grand
?C = cc    ?M = cmth5    ?Pdt2 = child    ?Rdt = parent
?C = cc    ?M = cmth5    ?Pdt2 = parent   ?Rdt = grand
?C = cc    ?M = cmth5    ?Pdt2 = parent   ?Rdt = parent
Yes.
```

It is important to check that any method of a defined object (i.e. class and structure instance) with a signature defined is included in one of the views just defined. In fact the union of **?C[METHODS->?M]** and **?C[INHMETHODS->?M]** are all of the properties that are covered by the views. The following denial query checks whether there exists a method, of a user defined object, that is not included in a view.

```
?- ?C:udo[ ?M*=>?_Dt ; ?M=>?_Dt ],
   not  (?C[methods->?M; inhmethods->?M] ;
         (?M=?Ms(?_P),?C[methods->?Ms; inhmethods->?Ms]) ).
No.
```

The converse denial, that is there a method included in a view's list but no signature is present in the user defined object instances, follows:

```
?- ?C:udo[methods->?M; inhmethods->?M],
   not (?C[?M*=>?_Dt;?M(?_P)*=>?_Dt, ?M=>?_Dt;?M(?_P)=>?_Dt]).
No.
```

This method names coverage needs to be complete because a number of data typing and inference techniques and procedures use these views repeatedly.

10.3 Method Signature's Cardinality

In earlier parts of our framework the cardinality constraints attached to properties in **CLASS** and **STRUCTURE** definitions were used extensively. In one area these were used to guide and enforce inter-object relationships. In this section we present ways to check if an object instance attributes and methods is respecting the data-type signature and the cardinality constraints associated with it.

Flora-2 does not implicitly check and enforce these although it does offer some methods to aid in explicitly doing these at runtime; these are found in its *typecheck* module. Also the manual states two important points [YANGG08]: firstly, "it is theoretically impossible to have a query that will flag a violation of a cardinality constraint if and only if one violation existss *and* will terminate"; secondly, relative Flora-2 methods "may trigger run-time errors if there are rules that use non-logical features or certain built-ins in their bodies"; (an example of built-ins are

arithmetic comparisons over integers). The first argument was presented in our earlier literature review on data type checking,

How can we check a property's cardinality limits? A generic procedure called **%TYPE_ERROR_CHECK** is invoked with two arguments: the first is the **CLASS** or **STRUCTURE** instance and the second the type of check required. The idea behind the procedure is to check for each method definition if there are any object instances that have an assignment outside the cardinality limits. The following invocation checks a selection of methods applicable to class instance **PERSON** for cardinality issues; specifically in inheritable methods. No issues are actually reported and the **FALSE** result aids in traversing all possible values.

```
?- %type_error_check(person,"cardinality const.s – inherit. scalar mthds").
No.
```

The workings of the procedure start by first extracting the list of inheritable methods from the appropriate view (e.g. **?C[INSCALARMETHODS->?M]**). The cardinality checks are actually done in recursive procedure **%CC_INHSCALAR** that takes four arguments. The first and second arguments are the **CLASS** instance and the respective method list while the last two are the start and return list of objects with cardinality outside the permitted range. According to the content of the return list the procedure **%TYPE_ERROR_CHECK** invocation prints results.

```
%type_error_check(?C,"cardinality constraints – inherit. scalar methods"):-
    ?C[inhscalarmethods->?Mlst], %cc_inhscalar(?C, ?Mlst,[], ?Broken),
    if not ( ?Broken = [] )
    then ( write('cardinality constraints - )@_prolog,
           write('inheritable scalar methods - broken list')@_prolog,
           write(?C)@_prolog, write(' - ')@_prolog, writeln(?Mlst)@_prolog,
           writeln(?Broken)@_prolog,
           false )
    else ( writeln('nothing broken!?')@_prolog,
           false ).

    %cc_inhscalar(?C,[?M|?Mlst],?Ilst,?Flst):-
        ?Lst=collectset
             {?I|Cardinality[%_check(?O[?M{?L:?H}=>?_])]@_typecheck,
                            ?O:?C,?C:udo,?I=[?C,?O,?M,'n/a',?L,?H]},
        if (?Lst=[])
        then (%cc_inhscalar(?C,?Mlst,?Ilst,?Flst))
        else (%cc_inhscalar(?C,?Mlst,[?Lst|?Ilst],?Flst)).
    %cc_inhscalar(?_C,[],?Flst,?Flst).
```

The recursion of procedure **%CC_INHSCALAR** is controlled by the second argument; specifically the method list. On each invocation the respective method name is passed to Flora-2 type check module to search for any instances which are outside the permitted range; the type check method is called **%_CHECK** and takes a signature as input; for inheritable methods it is **?O[?M{?L:?H}=>?_]**. The return values from **%_CHECK** are further filtered for instances of a

user-defined objects and are assigned to a list of tuples that is appended to the third argument of procedure. Each tuple appended to the list has details of specific infringement found. Recursion stops when no more methods are left (i.e. in second argument) and the final list is unified to the forth argument.

The following is a simple example of cardinality issues that are captured. Assume we have another **CLASS** instance, called **CHILD**, which is *ISA* related to **PARENT** (given earlier in this section). Also note the **CHILD** instances data assignments (i.e. **CO1, CO2,** and **C03**).

```
child:class.
child::parent

child [schoolat{1:1} *=> string, kids{1:1} *=> integer ].

co1:child [fname->"c1name", livesat->"c1home", schoolat->"c1school"].
co2:child [fname->{"c2name","c2n"}, livesat->"c2h", schoolat->"c2s",
           kids->2].
co3:child [fname->"c3name", livesat->"c3h", schoolat->"c3s",
           kids->0].
```

On invoking the procedure on **CLASS** instance **CHILD** one gets the following reporting related to cardinalities issues. The list of **CHILD** inheritable method is printed and then the lists of instances outside the acceptable values. The first list says method **KIDS** in instances **CO1** and **CO2** have their lower limit broken (but higher limit is 'ok'), and the second list says that upper limit of method **FNAME** is broken in instance **CO1**. These are easily verified with the above instances.

```
?- %type_error_check(child,"cardinality constraints-inherit. scalar mthds").
cardinality constraints - inheritable scalar methods - broken list
child - [fname,kids,livesat,schoolat]
[[[child,co1,kids,n/a,1,ok], [child,co2,kids,n/a,1,ok],
 [[child,co2,fname,n/a,ok,1]]]
No.
```

There are three other procedures to cater for cardinality constraints: namely inheritable methods with an argument, and non-inheritable methods with and without arguments. Other than the view to use there is only one other main difference between each; specifically it is the argument of the **%_CHECK** procedure invocation to match the methods and cardinalities of interest.

There are some technical issues to discuss. First and foremost cardinality checking comes at a computational cost which is significant. Second there is an issue with checking the lower cardinality constraint of methods with an argument; the check reports an error when there are none. Flora-2's maintainers confirmed this and at this point there seems little to do to address

it through the current **%_CHECK** procedure implementation (*pers. comm.* Kifer, M [2013]). Yet another issue is that evaluation of **%_CHECK** is disrupted when recursive data assignments are found (e.g. list of **CLASS** instances).

10.4 Scalar Method Data Type Checking

In F-logic method signatures and data expressions are enforced through its typing constraints. An important result is that of well typing of an F-logic program is un-decidable. In this logical language type checking takes more the role of data-type coverage.

In Flora-2 type denotations are read but type checking is not implicitly checked on loading and evaluation. As in the case of method's cardinality run-time checks for well typeness are possible if the program is equipped with data-type checking procedures that are invoked at run-time. Although Flora-2 offers a procedure to test type correctness this framework opted to implement its own procedure and object structures (e.g. rules for views). The main reason for opting out is to favour flexibility, coverage, and control of the data-typing regime that we implement. For example, Flora-2 does not type check input restriction and relaxation of methods.

The type-checking of a signature against objects is done through five high-level procedures; with the first two presented in this section, another two that data type check methods with an argument, and the last dedicated to type checking a list of objects (i.e. a polymorphic and recursive data structure).

The high-level call to **%TYPE_ERROR_CHECK** is supplemented with two arguments: the first is the user defined object to type check and the second is the string that qualifies which check to run (i.e. inheritable scalar methods). From the following script it transpires that **CLASS** instance **CHILD** has three attributes of interest: namely **FNAME**, **LIVESAT**, and **SCHOOLAT** all of which are strings. For each **CHILD** instance and attribute a list of values assigned are listed. It transpires that all values are of the type expected.

```
?- ?C=child ,
   %type_error_check(?C,"case - inhe. scalar methods with result poly.").
UDO / inh scalar method / data type list / result data list:
child
fname
[string]
[ _datatype(_string(c1name),_string), …
  _datatype(_string(c4name),_string)   ]
```

```
*
UDO / inh scalar method / data type list / result data list:
child
livesat
[string]
[ _datatype(_string(c1home),_string), …
  _datatype(_string(c4home),_string)   ]
*
UDO / inh scalar method / data type list / result data list:
child
schoolat
[string]
[ _datatype(_string(c1school),_string), …
  _datatype(_string(c4school),_string)]
*
?C = child
Yes.
```

If the following **CHILD** fact is added and the same type check is invoked then a type check error is expected as attribute **SCHOOLAT** is assigned an integer.

```
co9:child
[fname->"c9name"^^_string, livesat->"c9home"^^_string,
 schoolat->"9"^^_integer].
```

In fact this transpires in the following sterilised output. At this point semantic action is required to rectify data type error detected.

```
?- ?C=child ,
   %type_error_check(?C,"case – inhe. scalar methods with result poly.").
…
UDO / inh scalar method / data type list / result data list:
child
schoolat
[string]
[ _datatype(_string(c1school),_string, … ,
  _datatype(_datatype(_integer,[57]),_integer)]
*
--- data type error - inh scalar - class / scalar method / result data
child
schoolat
_datatype(_datatype(_integer,[57]),_integer)
*
?C = child
Yes.
```

The procedure **%TYPE_ERROR_CHECK** for an inheritable scalar method needs to compare two sets. The first set comprises possible data types that the method returns (e.g. there could be more than one incomparable data type because signatures allow method overloading of result type). The second set are objects that are return values of the current method being investigated (but only objects that are instances of the current class and not of any of its subclass). The comparison is a universal quantification: there is no element in the second set that is not an instance-of any element in the first set. Otherwise we have a data-type error between the method and the data value. The only added feature is to ensure that we can favourably compare objects that are instances of the current class or any of its super classes.

The procedure starts by retrieving all inheritable and scalar attributes of the **CLASS** instance through the appropriate view and then calls recursive procedure **%TEC_INHSCALARPOLY** with two arguments: namely the **CLASS** instance and its method list retrieved. **%TEC_INHSCALARPOLY** terminates when all methods are processed. In **%TEC_INHSCALARPOLY** the two lists are built one of instances that match the method and the other the return data types of the same method. These two lists are passed to another recursive procedure, called **%TEC_INHSCALARPOLY_DT_O** to compare the two sets. The comparison is done with another recursive procedure called **%TEC_INHSCALARPOLY_DT_O_CHECK** that compares an object with the list of possible data types. The comparison is broken down into two cases: the first is for basic domains and this requires a mapping for a basic domain into a Flora-2 domain and check for membership; and the second case caters for output relaxation of the instance-of. These cases are encoded in procedure **%TEC_OUTPUTRELAX**. The respective procedures follows:

```
%type_error_check(?C,"case - inher scalar methods with result poly.") :-
  ?C[inhscalarmethods->?Mlst], %tec_inhscalarpoly(?C,?Mlst).

  %tec_inhscalarpoly(?C,[?M|?Mlst]):-
    ?Rdtlst  = {?Rdt | ?C[?M*=>?Rdt],  not compound(?M)@_prolog},
    ?Ordalst = collectset{?Ro  | ?O:?C[?M->?Ro], not compound(?M)@_prolog,
                                 if (?Sc::?C) then (not ?O:?Sc[?M->?Ro]) },
    writeln('UDO / inh scalar method / data type lst / result data lst: '),
    writeln(?C)@_prolog,
    writeln(?M)@_prolog,
    writeln(?Rdtlst)@_prolog,
    writeln(?Ordalst)@_prolog,
    writeln('*')@_prolog,
    %tec_inhscalarpoly_dt_o(?C,?M,?Rdtlst,?Ordalst),
    %tec_inhscalarpoly(?C,?Mlst).
  %tec_inhscalarpoly(?_,[]).

    %tec_inhscalarpoly_dt_o(?C,?M,?Rdtlst,[?Orda|?Ordalst]) :-
      %tec_inhscalarpoly_dt_o_check(?C,?M,?Rdtlst,?Orda),
      %tec_inhscalarpoly_dt_o(?C,?M,?Rdtlst,?Ordalst).
    %tec_inhscalarpoly_dt_o(?_,?_,?_,[]).

      %tec_inhscalarpoly_dt_o_check(?C,?M,[?Rdt|?Rdtlst],?Od):-
        if ( %tec_outputrelax(?Od,?Rdt) )
        then ( true )
        else ( %tec_inhscalarpoly_dt_o_check(?C,?M,?Rdtlst,?Od) ).

      %tec_inhscalarpoly_dt_o_check(?C,?M,[],?Od) :-
        ?Odte = collectset{?O|?O:?C[?M->?Od],?Sc::?C,not ?O:?Sc[?M->?Od] },
        w('--- data type err. - inh scalar - class/scalar mth/result data'),
        writeln(?C)@_prolog,
        writeln(?Odte)@_prolog,
        writeln(?M)@_prolog,
        writeln(?Od)@_prolog,
        writeln('*')@_prolog.

      %tec_outputrelax(?O,?U):-
        ?U:udo, ?O:?U,
        if (?_Sc:udo, ?_Sc::?U, ?O:?_Sc ) then (false) else (!).
      %tec_outputrelax(?O,?U):- not ?U:udo, dt(?U,?Rdt1), ?O:?Rdt1, !.
```

As an example with output relaxation polymorphism for the result data type consider the following addition to the **CLASS** instance **CHILD**. The attribute **FAVINLAW** is denoted by a **PARENT** data type. The following objects have been assigned to the **CHILD** class and a type check invoked on it. The following scripts show the execution trail of the type check; specifically note the type check complaining about instance **CO1** when an instance-of **PARENT** or **GRAND** is expected.

```
child [favinlaw{1:1}*=>parent ].

co1:child [favinlaw-> go1].
co2:child [favinlaw-> po1].
co3:child [favinlaw-> po2].
co4:child [favinlaw-> co1].

?- ?C=child,
   %type_error_check(?C,"case – inhe. scalar methods with result poly.").
UDO / inh scalar method / data type list / result data list:
child
favinlaw
[grand,parent]
[co1,go1,po1,po2]
*
--- data type error - inh scalar - class / scalar method / result data
child
[]
favinlaw
co1
* …
?C = child
Yes.
```

To data type check for scalar and non-inheritable methods a procedure, very similar to the above exists.

10.5 Arity Method Data Type Checking

In this section we deal with methods that return data and also take an argument. (Without much loss of generality we are limiting the number of arguments to one). The procedure to type check such a method is very similar to the one adopted for scalar methods. A basic difference is that we need to type check the argument too rather than just the return type. Also the argument data type expected is substitutable by any subclass of the argument – i.e. input restriction.

The **%TYPE_ERROR_CHECK** procedure takes two arguments: the first is the class instance and the second is a string that identifies the check required. Once invoked the procedure retrieves a list of inheritable with arity methods and calls the recursive procedure **%TEC_INHARITYPOLY** with this list of methods. In this procedure two lists are generated: the

first is all possible type signatures that satisfy the current method by considering argument data type restriction and return data type relaxation (note that this is available to us in view **AS(…)**); the second is a list of tuples (where the each tuple holds the argument and return value for the current method definition). These two lists are passed to another recursive procedure, called **%TEC_INHARITYPOLY_DT_O**, to run a universal quantification between the two sets; namely there is not data tuple that is not an instance-of an argument and return data type tuple. If it is not the case we have established a type error and it is reported. To determine if two tuples from the respective list equate there are two procedures that cater for this. The first, called **%TEC_INHARITYPOLY_DT_O_CHECK** scans over the data items pair to every data type pair and the second procedure does the actual comparison between for each part of each pair.

```
%type_error_check(?C,"case-inher. mths with parameter and result poly")
  :-  ?C[inharitymethods->?Mlst], %tec_inharitypoly(?C,?Mlst).

  %tec_inharitypoly(?C,[?M|?Mlst]):-
    ?Rpdtlst  = collectset{?I1|as(?C,?M,?Pdt1,?Rdt1,"inh")[class->?C],
                             ?I1=[?Pdt1,?Rdt1]},
    ?Orpdalst = collectset{?I2|?O:?C[?M(?Po)->?Ro],
                             if  (?Sc::?C)
                             then (not ?O:?Sc[?M(?Po)->?Ro]),
                             ?I2=[?Po,?Ro]},
    w ('class / inher arity method / result & para. '),
    w ('data type list / result and parameter data list '),
    writeln(?C)@_prolog,
    writeln(?M)@_prolog,
    writeln(?Rpdtlst)@_prolog,
    writeln(?Orpdalst)@_prolog,
    writeln('*')@_prolog,
    %tec_inharitypoly_dt_o(?C,?M,?Rpdtlst,?Orpdalst),
    %tec_inharitypoly(?C,?Mlst).
  %tec_inharitypoly(?_,[]).

    %tec_inharitypoly_dt_o(?C,?M,?Rpdtlst,[?Orpda|?Orpdalst]):-
      %tec_inharitypoly_dt_o_check(?C,?M,?Rpdtlst,?Orpda),
      %tec_inharitypoly_dt_o(?C,?M,?Rpdtlst,?Orpdalst).
    %tec_inharitypoly_dt_o(?_,?_,?_,[]).

      %tec_inharitypoly_dt_o_check(?C,?M,[?Rpdt|?Rpdtlst],[?Op,?Od]):-
        ?Rpdt=[?Pdt,?Rdt],
        dt(?Pdt,?Pdt1),
        dt(?Rdt,?Rdt1),
        if    ( %tec_instanceofonly(?Op,?Pdt1),
                %tec_instanceofonly(?Od, ?Rdt1) )
        then ( true )
        else ( %tec_inharitypoly_dt_o_check(?C,?M,?Rpdtlst,[?Op,?Od])).

      %tec_inharitypoly_dt_o_check(?C,?M,[],[?Op,?Od]):-
        ?Odte = collectset{?O | ?O:?C[?M(?Op)->?Od], ?Sc::?C,
                                not ?O:?Sc[?M(?Op)->?Od] },
        w('--- data type error - inher poly arity ),
        w(' - class / method / parameter data / result data'),
        writeln(?C)@_prolog,
        writeln(?Odte)@_prolog,
        writeln(?M)@_prolog,
        writeln(?Op)@_prolog,
```

```
        writeln(?Od)@_prolog,
        writeln('*')@_prolog.

      %tec_instanceofonly(?I,?C) :-
        if   ( ?C:udo )
        then ( ?I:?C, if ( ?Sc::?C, ?I:?Sc ) then ( false ))
        else ( ?I:?C ).
```

As an example consider a **CLASS** instance called **CC** with one arity and inheritable method, called **CMTH5**, defined. A number of instances of **CLASS CC** are defined too. On running the type check on **CC** it transpires from the check that two type errors have been detected and therefore these need addressing.

```
cc:class[ cmth5(parent){1:1}*=>parent ].
  cco1:cc[ cmth5(po1)->co1 ].
  cco2:cc[ cmth5(go1)->po1 ].
  cco3:cc[ cmth5(po2)->po3 ].
  cco4:cc[ cmth5(co1)->go1 ].

?- ?C=cc,
   %type_error_check(?C,"case - inher mths with param. and result poly.").
…
class / inher arity method / result & parameter data type list / result and
parameter data list
cc
cmth5
[[child,grand],[child,parent],[parent,grand],[parent,parent]]
[[co1,go1],[go1,po1],[po1,co1],[po2,po3]]
*
--- data type error - inher poly arity - class / method / parameter data /
result data
cc
[]
cmth5
go1
po1
*
--- data type error - inher poly arity - class / method / parameter data /
result data
cc
[]
cmth5
po1
co1
*
?C = cc
Yes.
```

A similar procedure for arity methods, but non inheritable, is also available. The basic difference is the view that is used to generate the methods to type check.

10.6 Recursive Data Type Checking

The list data structure has been extensively used in our framework. In F-logic lists are used to create parameterised families of classes too by bounded polymorphism. Flora-2 supports these too. What follows are the general definitions required to integrate lists into our framework by having the possibility of creating a list of objects as a **CLASS** instance. (Most of the signature

definitions are adopted from F-logic paper [KIFER95] and Flora-2 manual [YANGG08]). A logical program that adopts the following definitions has to ensure that the program evaluation does not enter into infiniteness; i.e. the evaluation never terminates. Clearly this is a concern too as we want type checking to terminate.

The following rules help structure lists. If **?H** represent an instance-of an object (**?C**) and there exists a list of objects of **?C** then their concatenation is also a list of objects **?C**. The null list, i.e. **[]**, is by definition a list of objects **?C**. Also if two **CLASS** instances are related through the *ISA* relationship then so is the list of respective **CLASSES**.

```
[]:list(?C).
[?H|?Rlst]:list(?C) :-  ?H:?C,  ?Rlst:list(?C).
list(?T)::list(?S)  :-  ?T:udo, ?S:udo, ?T::?S.
```

The following are lists of objects that are instances of **CLASS**. For example list of integers is an instance-of a **CLASS** (e.g. **LIST(INTEGER)**).

```
list(integer):class.
list(integer)[ on{1:1} *=> Boolean ].

[1,2,3]:list(integer).  [1,2,3][on->true].
[4,5]:list(integer).    [4,5][on->true].
```

Examples of lists of a user defined object **PERSON** follows. Each list instance has in turn its instances (e.g. **[PO1, PO2, PO3]** is an instance-of **LIST(PERSON)**). Note the logical identifier of a list is its content; i.e. the identifier of the first list is **[PO1]**.

```
list(parent):class.
list(parent)[on{1:1}*=>boolean].

[po1]:list(parent).          [po1][on->true].
[po2]:list(parent).          [po2][on->true].
[po1,po2,po3]:list(parent). [po1,po2,po3][on->true].
[po1,go1]:list(parent).      [po1,go1][on->true].
[po1,co1]:list(parent).      [po1,co1][on->true].
```

The following is an example of a list of lists of a **CLASS** instance **PERSON**.

```
list(list(parent)):class.
list(list(parent))[on{1:1}*=>boolean].

[[po1]]:list(list(parent)).                  [[po1]][on->true].
[[po1],[po1,co1]]:list(list(parent)).        [[po1],[po1,co1]][on->true].
[[po1],[po1,go1]]:list(list(parent)).        [[po1],[po1,go1]][on->true].
```

Since the query **?I:LIST(PARENT)** is not computable due to infiniteness one can try to control evaluation from entering an infinite loop. There are a number of techniques: first one can control the engine that runs Flora-2 inference to stop on such evaluation patterns (i.e. XSB Prolog evaluation is tweaked); secondly one can ensure that each query avoids this evaluation branch by instantiating a list instance to a variable with a standard expression that does not

trigger a spiral behaviour (e.g. using **?I[ON->TRUE]** where this ground molecule has to be found in each list instance). For the second case the query becomes **?I[ON->TRUE], ?I:LIST(PARENT).** At this point of the research the latter is preferred. In the near future this is to be revisited as dedicated control of XSB evaluation for Flora-2 programs has been incorporated in a 2012 concurrent versioning system update of XSB Prolog distribution. (XSB and Flora-2 CVS are on SourceForge – { FLORA.SOURCEFORGE.NET }).

One can attach a signature and logic to the polymorphic **LIST(?)** object and through the instance-of relationship both signature and logic are inherited. Basic methods for lists include **FIRST**, **REST**, and **APPEND**. These and other definitions follow:

```
list(?T)[ first*=>?T, rest*=>list(?T),
            append(list(?T))*=>list(?T),
            lteinl(List(?T))*=>boolean,
            mininl(list(?T))*=>list(?T) ].

?Lst[first->?H, rest->?R] :-      // head and tail of list
    ?Lst:list(?_T),?Lst=[?H|?R].

?Lst[append(?Taillst)->?Newlst]:- // append a list to a list
   ?Taillst:list(?T),
   ?Lst:list(?T),
   ?Lst[_append(?Taillst)->?Newlst]@_basetype.

?Lst[lteinl(?Plst)->?BFlag] :-    // less than in length of list
    ?Lst:list(?T),
    ?Plst:list(?T),
    ?Lst[_length -> ?Slst]@_basetype,
    ?Plst[_length -> ?Splst]@_basetype,
    if ( ?Splst > ?Slst ) then ( ?BFlag=true ) else ( ?BFlag=false ).

?Lst[mininl(?Plst)->?Result] :-   //  which is the minimal (shortest) list
    ?Lst:list(?T),
    ?Plst:list(?T),
    if ( ?Lst[lteinl(?Plst)->true] )
    then ( ?Result=?Lst )
    else ( ?Result=?Plst ).
```

The following is an example of using polymorphic methods (e.g. **FIRST** and **MININL**) against list of **PARENTS**.

```
?- ?I[on->true],?I:list(parent),?I[first->?F].
?I = [co1, co2]   ?F = co1
?I = [po1]      ?F = po1
...
Yes.
?- ?I1[on->true], ?I1:list(parent),
   ?I2[on->true], ?I2:list(parent),
   ?I1[ mininl(?I2)->?Minlst ].
?I1 = [co1, co2]
?I2 = [co1, co2]
?Minlst = [co1, co2]
?I1 = [co1, co2]
?I2 = [po1]
?Minlst = [po1]
...
Yes.
```

In our type checking of scalar and arity methods the basic check was that an object is an instance-of any of the types indicated. Consequently lists, if properly asserted through an instance-of on the lines just presented, can be type checked. But there lies a caveat. The assertion that an object is an instance-of a **LIST(?C)** object does not imply in Flora-2, together with our **LIST(?C)** signatures, that all of its constituents are indeed an instance-of **?C**. To type check an expression which includes a list requires not only the checks mentioned earlier but also a check on the content of specific **LIST(?C)** instance. For this purpose another type of error-check procedure is introduced. Lists' being recursive structures in definition and content forces such a procedure to cope with list of list recursively. The following script shows an invocation on lists presented earlier on **LIST(PARENT)** and **LIST(LIST(PARENT))**. Note the type check flagging of infringements.

```
?- ?L=list(parent),%type_error_check(?L,"case - list(?T) instances are of
type ?T (or deep extent): ").
case - list(?T) instances are of type ?T (or deep extent):
flapply(list,parent)
[[co1,co2],[po1],[po1,co1],[po1,go1],[po1,po2,po3],[po2]]
tec_listof  true
--- data type error - case - list(?T) instances are of type ?T (or deep
extent):
flapply(list,parent)
parent
[po1,go1]
go1
*
?L = list(parent)
Yes.
?- %type_error_check(list(list(parent)),"case - list(?T) instances are of
type ?T (or deep extent): ").
case - list(?T) instances are of type ?T (or deep extent):
tec_listof  false
flapply(list,parent)
[[[po1]],[[po1],[po1,co1]],[[po1],[po1,go1]]]
--- tec_listof_recurse >>>>
…
--- data type error - case - list(?T) instances are of type ?T (or deep
extent):
flapply(list,parent)
parent
[po1,go1]
go1
*
Yes
```

The type-error check for testing the content of any recursive list is similar for those presented earlier except for the possibility that a list's elements are themselves a list (and possibly with more depth). The procedure invocation requires two arguments the first is a list-based structure (**?L**) and the second is the customary string to identify the task. The first task of the procedure is to check that the first argument is indeed a list and if so it builds a list of objects

that are instance-of the list structure (i.e. **?L**) and then passes **?L** and the list of its instances to procedure **%TEC_LISTOF**. This procedure has to check whether **?L**, when unified with **LIST(?DT)**, is defined through another list. If it is not the case and **?DT** is either a basic domain or an instance-of **CLASS** (but not of **LIST(?C)** nature) then procedure **%TEC_LISTOF_DT** is called with three arguments. The arguments are **?L**, **?DT**, and the **LIST(?)** instances. A universal quantification query checks if each **LIST(?)** instance element is an instance-of **?DT**. The remaining procedures are similar to the previous type error check own procedures from this point on.

If the initial check in **%TEC_LISTOF ?L** unifies with **LIST(LIST(?_DDT))** then basically we need to call another recursive branch of **%TEC_LISTOF** through procedure **%TEC_LISTIF_RECURSE**. The later has to be recursive to ensure that depth of list within a list is catered for.

```
%type_error_check(?L,"case - list(?T) inst are of type ?T (or deep extent):
"):-
  w('case - list(?T) instances are of type ?T (or deep extent): '),
  if ( ?L = list(?_Dt) )
  then ( ?Ilst=collectset{?I|?I[on->true],?I:?L,?L:udo},
         %tec_listof(?L,?Ilst))
  else ( w('--- data type error - arg. provided is not a list(?T) type!'),
         writeln(?L)@_prolog,
         fail).

  %tec_listof(?L,?Ilst) :-
    if   ( ?L = list(?Dt),
           (?Dt:udo;?Dt=integer;?Dt=string),
           not ?Dt=list(?_Ddt) )
    then ( writeln(?L)@_prolog,
           writeln(?Ilst)@_prolog,
           %tec_listof_dt(?L,?Dt,?Ilst) )
    else ( if   ( ?L = list(?Dt2), ?Dt2=list(?_Ddt) )
           then ( writeln(?Dt2)@_prolog,
                  writeln(?Ilst)@_prolog,
                  %tec_listof_recurse(?Dt2,?Ilst))
           else ( w('--- data type error `),
                  w(`- arg. provided is not a list(?T)'),
                  writeln(?L)@_prolog,
                  fail) ).

     %tec_listof_recurse(?Dt,[?Hl|?RIlst]):-
        writeln('--- tec_listof_recurse >>>>')@_prolog,
        writeln(?Dt)@_prolog,
        writeln(?Hl)@_prolog,
        %tec_listof(?Dt,?Hl),
        %tec_listof_recurse(?Dt,?RIlst).
     %tec_listof_recurse(?_Dt,[]):- !.

        %tec_listof_dt(?L,?Dt,[?Lst|?Ilst]):-
          %tec_listdt_dt_o(?L,?Dt,?Lst,?Lst),
          %tec_listof_dt(?L,?Dt,?Ilst).
       %tec_listof_dt(?_L,?_Dt,[]) :- !.

          %tec_listdt_dt_o(?L,?Dt,?OLst,[?H|?Lst]):-
            if   (%tec_listoutputrelax(?H,?Dt))
```

```
            then (%tec_listdt_dt_o(?L,?Dt,?OLst,?Lst))
            else ( w('--- data type error - case - list(?T)'),
                   writeln(?L)@_prolog,
                   writeln(?Dt)@_prolog,
                   writeln(?OLst)@_prolog,
                   writeln(?H)@_prolog,
                   writeln('*')@_prolog ).
          %tec_listdt_dt_o(?_L,?_Dt,?_Lst,[]):- !.
```

10.7 F-Bounded Polymorphism

In the preceding sections we have seen how objects having a data-type signature and *ISA* relationship assertions are type checked, during run time, and where checking can identify a good range of data-typing errors. The type checking dealt with attributes data type, arity methods with variance and contra-variance data type, and recursive structures (i.e. list).

In our literature review it was stated that a subset of universally quantified polymorphism, i.e. bounded quantification, does not properly type check in the presence of recursive types. In the same review a solution was indicated and it is called F-bounded quantification.

To check if any user-defined object instance that has a type signature with self-reference one uses the following query. The pattern is looking for any **UDO** instances that have a signature component which is the same **UDO** instance; the query is restricted to inheritable methods only.

```
?- ?Clst=collectset{?C|?C:udo,(?C[?M*=>?C];?C[?M(?C)*=>?Rt])}.
?Clst = [po, list(integer), list(parent), list(pt), list(list(parent))]
…
Yes.
```

If one wants to identify recursive methods of classes (point (**PT**) and partial order (**PO**)) then following query will oblige:

```
?- ?C=list(pt),?C[?M*=>?C;?M(?C)*=>?Rt;?M(?Pt)*=>?C].
…
?C = list(pt)    ?M = append    ?Rt = ?_h5642    ?Pt = list(pt)
?C = list(pt)    ?M = append    ?Rt = list(pt)   ?Pt = ?_h5594
?C = list(pt)    ?M = rest      ?Rt = ?_h5499    ?Pt = ?_h5504
…
Yes.

?- ?C=po,?C[?M*=>?C;?M(?C)*=>?Rt;?M(?Pt)*=>?C].
?C = po   ?M = leqinl   ?Rt = Boolean   ?Pt = ?_h5165
?C = po   ?M = min      ?Rt = ?_h5204   ?Pt = po
?C = po   ?M = min      ?Rt = po        ?Pt = ?_h5187
?C = po   ?M = min(po)  ?Rt = ?_h5138   ?Pt = ?_h5143
Yes
```

If **CLASS** instance partial order (**PO**) has a sub-class called number (**NB**) then we expect the latter **LEQINL** method to have **LEQINL(NB)*=>BOOLEAN** rather than **LEQINL(PO)*=>BOOLEAN** to fit the contra-variance between these functional methods. To get this pattern the **CLASS** instances **PO** and **NB** need to be re-cast as follows. First create a

parametric class instance with data signature (i.e. **FB_PO(?T)**), then create a parametric class instance with a data type instantiating the **?T** variable for each type required, (e.g. **FB_PO(PO)**). Then the substituted types are in an *ISA* relationship this is reflected in the data signature of each respective parametric class instance. Consequently, is the method **LEQINL** in the right subtype order?

```
fb_po(?T)[leqinl(?T)*=>Boolean].

fb_po(po):class.
fb_po(nb):class.

po:class.
nb:class.

nb::po.

?- fb_po(?T)[?M*=>?Rt].
?T = ?_h2514      ?M = leqinl(?_h2514)     ?Rt = Boolean
Yes.

?- fb_po(po)[?M*=>?Rt].
?M = leqinl(po)   ?Rt = boolean
Yes.

?- fb_po(nb)[?M*=>?Rt].
?M = leqinl(nb)   ?Rt = boolean
Yes.
```

At this point, where the designer needs to revise the class structure (and possibly migrate objects from a **CLASS** instance to another) the following query helps to establish which artefacts depend on a "deprecated" class:

```
?- ?Deprecated=list(pt),?C[?M*=>?Deprecated;?M(?Deprecated)*=>?_Dt].
…
?Deprecated = list(pt)     ?C = fb_po(list(pt))     ?M = leqinl
Yes.
```

10.8 Data Type Checking & Inference in our Framework

In the previous sections we have been type checking the state of our object base. Before that we presented a number of integrity-constraint specifications and enforcement procedures that maintain a consistent object base. Nonetheless there are still possibilities that type errors are present. There are three main sources: first, our type system is not complete and neither completely adequate; second, the framework procedures themselves can develop type errors; and thirdly, think there are rules in the object base that have not fired because these are not satisfied within an object state.

Another important activity undertaken here is the building of new data types through type inference and identifying and applying fixes to known data types (e.g. F-bounded fix in

recursive data types). This has to be done post translation as EERM diagrams no not indicate constructs that map to parametric classes.

Nonetheless to attenuate the problems with type checking it must be noted that rules and procedures make extensive use of meta-programming that in turn type restricts the firing of rules. The expense of run-time checking and data-type inference is a useful technique for developing software even if strong typing is not adopted.

10.9 Summary

F-logic allows its objects to be associated with a variety of data type signatures and its evaluation is affected by type inference and its relationships (e.g. *ISA* and instance-of). But we have seen how Flora-2 does not implement these type checking and inference in its evaluation. In the context of deductive systems we develop type checking that caters for methods that are, static, arity methods, polymorphic (e.g. overloaded and bounded), and recursive structures (e.g. lists). All type checking and type inference is done through coding with Flora-2.

Flora-2 procedures type check a user-defined object instances' attributes, methods with one argument, and recursive data type (e.g. lists). These checks are based on making sure an object is covered by a type signature, and is done at run-time. Any type errors found are listed and explained. Other than specific type-checking procedures much of the code, presented in earlier sections, makes extensive use of signature expression when executing rules and procedures – i.e. it is less prone to data type errors because objects are filtered. This also makes writing software somewhat more disciplined.

The type checking also data checks properties overloading, parametric classes, and bounded polymorphism. The framework also identifies signatures that require F-bounded polymorphism transformation and suggest possible solutions. Also views are available to enumerate other object signatures that depend on a class that needs this transformation.

The type system implemented is not complete; the scope of the data-type checking is instances for classes and structures; secondly, only single argument arity methods are checked. This regime, as expected, is computationally expensive. Also some expressions can easily take an evaluation into an infinite process; e.g. queries related to lists.

Chapter 11

Object-Oriented Query Model

(I)

11 – Object-Oriented Query Model (I)

This chapter and the next introduce query modelling over our object database and its supporting framework. Query modelling is the design of language constructs that specify retrieval over an object base. A query specification includes a range, properties that any output must satisfy, and the structure of its output. It then needs to be processed and results made available.

Query modelling aspects presented in this chapter and the next include: instantiation of queries and their results; introduction of an object algebra as a query language to manipulate an object base; working out the data type of a query result; and a logic programming implementation of the algebra's semantics.

Our object algebra has the following characteristics: firstly, it is totally procedural in contrast to declarative constructs presented so far; secondly, its input are objects, including schema objects like data-type signatures, and its outputs are also objects (i.e. with logical identifier, value, data type signature); thirdly, it is value and identifier based; fourthly, its semantics are given in Flora-2 with important parts in declarative constructs. Another interesting characteristics is that the operators, unlike other algebras proposed, do not overlap much in their functionality.

The central theme here is our object algebra and its close coupling with data-modelling constructs. What justification exists for our algebra when a declarative system is present and workable? Firstly, database algebras are procedural and are optimisable. Secondly, with each algebraic query one can determine the correctness of the query and associate each query with a subset of declarative queries. Thirdly, if the size of the data is greater than available RAM then procedural techniques based on storage access are required. Furthermore algebra aids in our understanding of complex queries, and is an excellent teaching tool in itself.

There are a good number of algebraic operators and many have their origin in the relational algebra. These include 'union', 'difference', 'product', 'project', and 'select'; their implementation and integration in our data model necessitate a wider semantics due to the

data model's rich constructs. Other operators include 'nest', 'unnest', 'rename', 'aggregate', and 'map' (and are described in the next chapter). Our algebra does have a basis in other proposals, see section 6.2 in chapter six, but has a number of additions and characteristics which makes it a distinct effort.

11.1 Basic Query Modelling

The traditional scope of query modelling is the retrieval, update, creation, and purging of objects from a database. Here we focus on the retrieval from an object base; nonetheless retrieval is part of other aspects of query modelling. For example it is a common operation to retrieve a set of objects and then purge these.

Retrieval of objects in an object-oriented data model is based both on navigation (i.e. logical identifier traversal) and on matching values that make up an object's value. Also the retrieval specifications have a high-level nature. Another important characteristic is that the query constructs cover a good portion of known query expression types; for a start a query model expresses all the queries of Codd's relational language – technically called Codd's completness.

Another of Codd's influence on query modelling is having both a declarative and a procedural language. Examples of declarative queries, found in this study, include F-logic and ODMG OQL queries. Ideally these languages should firstly be *equipollent*, and secondly, there should be a straightforward conversion from the declarative to the procedural language. Once a procedural construct is at hand one has a basis for query optimisation.

A query language is *closed* when a query's output can be input to another query. The relational model is a closed query language. Open query languages do not allow the output of the first query to be the input of a second. There are practical advantages for closed languages: firstly, one can write a cascade of queries; secondly, one can build dependences in composite query evaluation. There is another issue concerning the output of a query: what data-type signature should it take up. There are some possibilities for the output's data-type signature: it could be a set of identifiers; or a relation (i.e. like the relational

model); or a new object structure built from the input objects and the type of query operator. The output could also be a homogenous or heterogeneous collection.

If the queries themselves and the objects created from processing a query are objects in the framework too, then another level of object homogeneity has been achieved. Furthermore, a query language should integrate with object-oriented themes such as instance-of, integrity constraints, and data-type signatures. When integrating with an object-oriented paradigm a difficult issue is dealing with method invocations that change the state of an object during the processing of a query construct.

11.2 Object Algebra and an Object Framework (I)

This section presents an algebra we have purposely developed for our framework. We want this algebra to be the target of a subset of OQL queries, and also be able to query process and optimise the algebraic expressions. The set of operators are, collectively unique, and are tightly coupled with the underlying data model. An important consideration is that these algebraic operators are implemented with declarative constructs to describe their semantics. Another unique feature of the operators here is their capability to carry integrity constraints from the operator's ranges to the query output, if applicable.

We start with a short requirements specification for an object algebra to implement a query model. Therefore the algebra complements the framework and object-database data modelling found in the previous chapters.

The algebraic operators are to handle objects and object-oriented themes found in the framework as much as possible. The algebraic operators have to allow both value-based and logical-identifier based access methods. A number of the operators have relational data model origin, some to build object relationships, and some to unpack an object from its composite parts. In terms of ranges (of objects) for the algebraic operands these can be classes, structures, and query instances.

In terms of expressibility the algebra is to be at least Codd complete and offer a wide overlap with ODMG OQL. Furthermore the object algebra's output is closed; in the sense it creates objects (with identity, instance-of, and composite object).

In this exercise method invocations that change the state of an object during the processing of a query constructs are not acceptable. Since we are dealing with retrieval it is assumed that such methods are not within the scope of the study. Also the processing of a cascade of queries assumes that the underlying object base remains.

The algebra too requires its development through a logic programming language; i.e. Flora-2. The object algebra is to have run-time and interactive capabilities in aspects of query specification, query processing, and data-type inference of the results.

11.2.1 Our Algebra

The algebra comes with ten operators: union, difference, project, product, rename, select, map, aggregate, nest, and unnest. The operators work on an object-oriented collection. They have been described as value-based because the operands are cognisant of an object's state value, class instantiation, and its logical identifier. The operators have a common template for implementation: in general it is pre-condition satisfaction, instantiating the query instance and its details, instantiation query instance data type signature, implementation of the semantics, instantiating the result as instance-of the query at hand, and post check satisfaction (e.g. duplicate elimination). The framework supports the algebra with an array of methods that are shared across operators' implementation, and objects associated to each algebraic operator (e.g. object identified by **`WVF`** for map operator).

It has to be noted that the algebraic operators are tightly coupled with the framework, the data-model encoding, and a schema definition. For example, the range of an operand is either the deep extent of a class instance or limited to its immediate extent. Another example, is the data type checking and inference follows that which has been done in an earlier chapter. An important observation is that the operators are "shallow" in that they consider the top most structure of a range; the exceptions being select and map operators. The algebraic objects make extensive use of the framework as each query expression has an instance which is adorned by the query's details.

The input of an algebraic expression, as regards its range, is technically any extent of schema artefacts. In our framework the most evident ranges are instances of **`CLASS`**, **`STRUCTURE`**, and **`ALGQUERY`**. In chapter ten we have seen how parameterised classes of

lists were instances of **CLASS** or **STRUCTURE**; consequently a collection of lists are a range to these queries (albeit this requires more control in our coding when enumerating a query's range extent). The operand only works on the properties of ranges that have a data-type signature; therefore method overloading is catered for. The map operator requires a "side" input of the specification of the function to apply; the end user has the possibility of supplying it at run time. No operator is meant to invoke methods that change the input range while the query is being evaluated.

The output of an algebraic operand is set of objects whose state structure is a tuple and which is covered by data-type signatures. The tuple's data-type signatures are derived from the operands input ranges. Also the query output includes an assertion that the query is an instance of **ALGQUERY** object, and an assertion that the query results are an instance of the query instance. As the outputs of an expression are objects, and part of the framework, this implies that our algebra has the property of closure.

11.2.2 Object Algebra Constructs and their Realisation in the Framework

A dedicated object, identified as **ALGEBRA**, holds a number of procedures related to our object algebra, for example each operator has a dedicated procedure attached to it. Also for each algebraic operator an object is instantiated that contains reified rules required by that operator. These reified rules cover query instantiation, query-data type signature, generation of logical identifier for a query's instances, and materialising the data for the query's instances. For example the procedure for the operator union by value, **%UV**, is specified in **ALGEBRA[%UV(?C1,?C2,?PARAM)]** and takes three arguments – denoted by variables **?C1**, **?C2**, and **?PARAM**. The reified rules associated with it are found in object **UVF** with template **UVF[MTH(…) -> ${ … }]**.

Each algebraic query execution generates its objects (i.e. that satisfy it). At a *meta* level each query is made an instance-of **ALGQUERY**; much like classes were instances of **CLASS** and **STRUCTURE** in an Chapter eight. Each query needs an identifier whose composition follows the following template: **OPERATOR([ARG]+)**. For example the assertion **UV(CLASS1,CLASS2,"DEEPEXTENT"):ALGQUERY** states that we have an algebraic query

based on union by value between classes 1 and 2; the third argument is a parameter. Instances-of **UV(…)**, i.e. the query, are identified by the following assertion:

```
iuv(class1,class2,"deepextent",objectN,1):uv(class1,class2,"deepextent").
```

Also each **UV(…)** has a data signature associated with it. This is inferred from the classes 1 and 2 instances. Consequently it is expected that the input are covered by a data-type signature, and data type checked. Three such constructs are instances-of **CLASS**, **STRUCTURE**, and **ALGQUERY**. The following scripts shows three queries: the first query interrogates the object base for instance of **ALGQUERY**; the second query works out how many instances have been created for each **ALGQUERY** instance; and the third show instances of one particular **ALGQUERY** instance.

```
?- ?Q:algquery.
?Q = uv(person,postgrad,"extent")
?Q = uv(student,exstudent,"deepextent")
?Q = uv(student,exstudent,"extent")
Yes.

?- ?Qi=count{?I[?Q]|?Q:algquery,?I:?Q}.
?Qi = 5    ?Q = uv(person,postgrad,"extent")
?Qi = 9    ?Q = uv(student,exstudent,"extent")
?Qi = 10   ?Q = uv(student,exstudent,"deepextent")
Yes.

?- ?I:uv(person,postgrad,"extent").
?I = iuv(person,postgrad,"extent",dri,1)
…
?I = iuv(person,postgrad,"extent",susan,2)
Yes.
```

Database algebras are sometimes classified by the number of ranges they have in their input. In our algebra the operators are either *unary* (i.e. take a single range) or *binary* (i.e. take two ranges). Some operators take a name of a function as an argument but by "habit" these are ignored when classifying by number of ranges.

There are a large number of methods that are used across all operators. For example since the operators are value based then duplicate detection and elimination is required; consequently the methods **%DUPLICATES(…)** and **%DELDUPLICATE(…)** of object **ALGEBRA** are accessible across the framework.

11.2.3 Union and Difference

Union and difference operators are binary operators and are set theoretic in meaning. In this framework we insist that the operands are union compatible; i.e. the operators take

homogeneous and comparable lists of objects. The output of the operators is also data-type homogeneous and it is inferred from the operands' data type signatures. Also these two operators are adequate to derive other set operators, for example intersection.

Each of these operators has algebraic properties that are given in each respective sub-section.

11.2.3.1 Union Compatible

The idea of union compatibility in the relational data model is that two operands in an algebraic operation have the same arity and domains match one to one between the operand's attributes. Attribute names are indispensable. Not all algebraic operators require their operands to be union compatible.

In our object-oriented data and query model we have to develop this to cater for operands that are in a sub-class relation (i.e. *ISA*), and for the possibility that attributes take arguments. Also property names are used but generality is not lost as a renaming algebraic operation is available in this query model (see later section).

The object **ALGEBRA** has a procedure called **%UNIONCOMP** that takes two arguments and returns a result string. The procedure takes care of four cases through the invocation of procedure **%UC**. The first deals with the spurious case when two input arguments are identical. The second and third cases deal with arguments being in an explicit *ISA* relationship. For the fourth case another procedure is called, named **%UCT**, that checks that there is no method data signature in the first argument that is not in the second argument, and conversely there is no method in the second argument that is not in the first. The method's data type compatibility is done on name, cardinality constraints, and data type of a method argument and return data type. This implementation of union compatibility between two object data signatures does not determine proper implicit subtype relationship between two operands.

```
algebra[%unioncomp(?C1,?C2,?Text)]:-
  if (%uc(?C1,?C2,?Text))
  then ( writeln('union compatible')@_prolog, true )
  else ( writeln('not union compatible')@_prolog, false ).

  %uc(?C1,?C2,?Text) :- ?C1=?C2, ?Text="UC: Self",!.
  %uc(?C1,?C2,?Text) :- ?C1::?C2,?Text="UC: $1 ISA $2 related",!.
  %uc(?C1,?C2,?Text) :- ?C2::?C1,?Text="UC: $2 ISA $1 related",!.
```

```
%uc(?C1,?C2,?Text) :-
  if   (%uct(?C1,?C2),%uct(?C2,?C1))
  then (false)
  else (?Text="UC: all properties match",true).

  %uct(?C1,?C2) :- ?C1[?M{?B:?T}*=>?D], \+ ?C2[?M{?B:?T}*=>?D].
  %uct(?C1,?C2) :- ?C2[?M{?B:?T}*=>?D], \+ ?C1[?M{?B:?T}*=>?D].
  %uct(?C1,?C2) :- ?C1[?M{?B:?T}=>?D],  \+ ?C2[?M{?B:?T}=>?D].
  %uct(?C1,?C2) :- ?C2[?M{?B:?T}=>?D],  \+ ?C1[?M{?B:?T}=>?D].
```

The following interactive script shows the use of union compatibility. The first deals with checking union compatibility with itself. The second call confirms that **STUDENT** (i.e. second argument) *ISA* **PERSON**. The third check confirms that **EXSTUDENT** and **STUDENT** are union compatible even though they are not in an *ISA* relationship.

```
?- algebra[%unioncomp(person,person,?T)].
union compatible
?T = "UC: Self"
Yes.
?- student::person.
Yes.
?- algebra[%unioncomp(person,student,?T)].
union compatible
?T = "UC: $2 ISA $1 related"
Yes.
?- exstudent::student.
No.
?- student::exstudent.
No.
?- algebra[%unioncomp(exstudent,student,?T)].
union compatible
?T = "UC: all properties match"
Yes.
```

11.2.3.2 Union

The binary union algebraic operator meaning for union compatible arguments is straightforward. Any object that is an instance-of either argument is included in the result. Also the data type signature of the output is determined by a data type signature relationship of input arguments. There are two issues specific to our data and query model: the first is whether the extent or deep extent of the arguments is required; the second being the demands that this operator is value based and produces a set (i.e. requires duplicate check and elimination).

In the first example a spurious union between instances of **CLASS PERSON** is requested. The operator is conditioned to query only the extent of **CLASS PERSON**. The verbose output confirms the union compatibility and indicates correctly that duplicates are present. The verbose confirm duplicate deletion from the output too. The query's meta instance (i.e. instance-of **ALGQUERY**) and query result are shown in the script below (e.g. instances-of

UV(PERSON,PERSON,"EXTENT"), the verbose output of the algebraic operation, are shown too).

```
?- algebra[%uv(person,person,"extent")].
union compatible
UC: ok
UC Self
algquery instance created
signatures created
data loid and move done
duplicate
list[flapply(iuv,person,person,joe,1),flapply(iuv,person,person,jv,1),fla
pply(iuv,person,person,jv,1)]
duplicate deletion done
Yes.
?- ?Q:algquery,?I:?Q.
?Q = uv(person,person,"extent")     ?I =
iuv(person,person,"extent",dri,1)
?Q = uv(person,person,"extent")     ?I = iuv(person,person,"extent",j,1)
?Q = uv(person,person,"extent")     ?I = iuv(person,person,"extent",p1,1)
?Q = uv(person,person,"extent")     ?I = iuv(person,person,"extent",p2,1)
Yes.
?- ?I:person, not ?I:student.
?I = dri
?I = j
?I = joe
?I = jv
?I = p1
?I = p2
Yes.
```

In the following example a union between **CLASS** instance **STUDENT** and **EXSTUDENT** is requested. The operation is adorned with a deep extent directive. The verbose notes that arguments are union compatible as their properties match and no duplicates are detected on computation. The result includes all instances coming from **CLASSES EXSTUDENT**, and **STUDENT**, (and **POSTGRAD** by *ISA* inference).

```
?- algebra[%uv(student,exstudent,"deepextent")].
union compatible
UC: ok
UC: all properties match
algquery instance created
signatures created
data loid and move done
duplicate list[]
Yes.
?- ?Q:algquery,?I:?Q.
?Q=uv(student,exstudent,"deep…") ?I=iuv(student,exstudent,"deep…",mary,1)
…
?Q=uv(student,exstudent,"deep…")
?I=iuv(student,exstudent,"deep…",susan,1)
Yes.?- ?I:postgrad.
?I = susan
Yes.
```

For the union operator implementation two reified rules are used. These are attached to an object identified with **UVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert a **UV(…)** object is an instance-of **ALGQUERY**. The method takes four arguments: the first two are the collections on which the union is to be executed. The third contains an indicator to state whether a deep extent or an extent is required for the result. The forth contains the mode of union compatibility if indeed the first two arguments are compatible.

```
uvf  [algqueryinstance(?C1,?C2,?Param1,?Text) ->
  ${ (  uv(?C1,?C2,?Param1):algquery[ uctext->?Text ] :- true  ) }].
```

Therefore for the above second example this reification invocation produces the following fact:

```
uv(student,exstudent,"deepextent"):algquery[uctext->"UC: all prop.
match"].
```

The second reification concerns the data-type signature of the result. This is actually related to the union compatibility result. For example if its two arguments are the same then the data signature of the result is the same too. Also if all properties match then the result data-type signature is the same too. If the two arguments are in an *ISA* relationship then the data-type signature is of the ancestral one. In all cases the deep-extent directive does not affect these inferences; i.e. data-type signature of the output is of the data type signature of the arguments and not their descendants.

The **UVF** object uses the **ALGQUERYDT** property and it takes four parameters: the first two denote the collections to union, the third for extent selection, and the forth the union compatible string. The **ALGQUERYDT** has four instantiations each related to a different union compatible result string. In each instantiation four rules are inserted into the framework to cater for inheritable (i.e. ***=>**) and non-inheritable (i.e. **=>**) methods, and arity and non-arity (i.e. **NOT COMPOUND(?M)**) methods. Each rule has the same pattern but each fire for a particular data signature type. If there is a method signature in an argument class then it needs to be asserted in the output data type signature (i.e. of **UV(…)** object). Actually there are two generic reification patterns: the first deals with 'self' and 'all properties match', and the second deals with *ISA* relationship (i.e. whether **C1::C2** or **C2::C1**).

```
uvf[
  algquerydt(?C1,?C2,?Param1,"UC: Self") ->
  ${ (uv(?C1,?C2,?Param1)[?M{?B:?T}*=>?D]
        :- ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog),
     (uv(?C1,?C2,?Param1)[?M{?B:?T}=>?D]
        :-?C1[?M{?B:?T}=>?D],  not compound(?M)@_prolog),
     (uv(?C1,?C2,?Param1)[?M(?P){?B:?T}*=>?D]
        :-?C1[?M(?P){?B:?T}*=>?D]),
     (uv(?C1,?C2,?Param1)[?M(?P){?B:?T}=>?D]
        :-?C1[?M(?P){?B:?T}=>?D] ) },

  algquerydt(?C1,?C2,?Param1,"UC: $1 ISA $2 related") ->
  ${ (uv(?C1,?C2,?Param1)[?M{?B:?T}*=>?D]
        :- ?C2[?M{?B:?T}*=>?D], not compound(?M)@_prolog),
     (uv(?C1,?C2,?Param1)[?M{?B:?T}=>?D]
        :- ?C2[?M{?B:?T}=>?D],  not compound(?M)@_prolog ),
     (uv(?C1,?C2,?Param1)[?M(?P){?B:?T}*=>?D]
        :-?C2[?M(?P){?B:?T}*=>?D]),
     (uv(?C1,?C2,?Param1)[?M(?P){?B:?T}=>?D]
        :-?C2[?M(?P){?B:?T}=>?D] )  },

  algquerydt(?C1,?C2,?Param1,"UC: $2 ISA $1 related") -> ${ … },

  algquerydt(?C1,?C2,?Param1,"UC: all properties match") -> ${ … }
].
```

In the following query example, where classes have union compatibility on a common data type signature, the data-type signature of the operation's result is shown.

```
?- uv(student,exstudent,"deepextent")[?M*=>?Dt;?M=>?Dt].
?M = enrolon         ?Dt = course
?M = fname           ?Dt = string
?M = stage           ?Dt = string
?M = telno           ?Dt = integer
?M = result(string)  ?Dt = unit
Yes.
```

The reification of **ALGQUERYDT** for the above query creates the following rules that in turn create the data type signatures:

```
  uv(student,exstudent,"UC: all properties match")[?M{?B:?T}*=>?D]
    :- student[?M{?B:?T}*=>?D], not compound(?M)@_prolog).
  uv(student,exstudent,"UC: all properties match")[?M{?B:?T}=>?D]
    :- student[?M{?B:?T}=>?D],  not compound(?M)@_prolog.
  uv(student,exstudent,"UC: all properties match")[?M(?P){?B:?T}*=>?D]
    :-? student[?M(?P){?B:?T}*=>?D].
  uv(student,exstudent,"UC: all properties match")[?M(?P){?B:?T}=>?D]
    :- student[?M(?P){?B:?T}=>?D].
```

The actual evaluation of union by value is done through procedure **%UV** in object **ALGEBRA**. The procedure **%UV** takes three arguments: the first two are the classes' whose instances are to union (i.e. unified to variables **?C1** and **?C2**) and the third is an indicator on the extent required (i.e. unified to variable **?DEEP**). The first task is to determine if the input operands are union compatible by calling procedure **%UNIONCOMP**. If successful its third argument that denotes the mode of union compatibility is unified to variable **?TEXT**. The next step is to create a union by value instance (i.e. **UV(…)** as an instance-of **ALGQUERY**), this is done through reification of **ALGQUERYINSTANCE** (and as explained earlier) and then

inserting these rules into the object base (i.e. Flora-2 **INSERTRULE** predicate). To the object just created, i.e. **UV(…)**, the data type signature is assigned through the reification of rules found in the property **ALGQUERYDT** and qualified by the mode of union compatibility (i.e. **?DEEP**). The rules created need to be inserted in the object base too. After instantiating the query's meta structures it is time to materialise the instances of the query and this is done through three steps. The first ensures that the output's object has an identifier and is an instance-of the query instance (i.e. **IUV(…):UV(…)**)–procedure **%UVDATALOID** is called. The second ensures that every object instance-of **UV(…)** just created has its properties assigned values from the argument's instances.

The procedure **%UVDATA** covers this. The last step requires that the result is checked for value duplicates and if any are present then any extra instance is deleted from the framework object base – object **ALGEBRA** procedures **DUPLICATES** and **DELDUPLICATE** (these are explained in section 11.2.2.4 below).

```
algebra[%uv(?C1,?C2,?Deep)] :-
  algebra[%unioncomp(?C1,?C2,?Text)],
  writeln('UC: ok')@_prolog,
  writeln(?Text)@_prolog,
  uvf[ algqueryinstance(?C1,?C2,?Deep,?Text) -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  uvf[ algquerydt(?C1,?C2,?Deep,?Text)-> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  %uvdataloid(?C1,?C2,?Deep),
  writeln('data loid done')@_prolog,
  %uvdata(?C1,?C2,?Deep),
  writeln('move done')@_prolog,
  algebra[%duplicates(uv(?C1,?C2,?Deep),?Duplst)],
  write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
  algebra[%delduplicate(?Duplst,uv(?C1,?C2,?Deep))],
  writeln('duplicate deletion done')@_prolog.
```

The procedure **%UVDATALOID**, listed below, takes three arguments which are unified in the calling procedure **%UV**. Basically the procedure needs to create an identifier for each instance-of related to the procedure's first two arguments. For this the **COLLECTSET** predicate is used. Because the operator has control on what extent to include in the output this is reflected in the qualifier inside each **COLLECTSET** invocation. Since F-logic asserts the deep extent by inference than the 'extent' directive needs to weed off instances of any of its sub classes. Once the required **COLLECTSET** is bound to each **CLASS** instance then each set is passed on its respective procedure to create the **IUV(…)** identifiers; these are called

%UVINSERTLEFT and **%UVINSERTRIGHT**. Actually these procedures are very similar to each other and only differ in the composition of the logical identifier created.

```
%uvdataloid(?C1,?C2,?Deep) :-
  if   (?Deep="extent")
  then (?Ileftlst=collectset{?Ileft|  ?Ileft:?C1,
                                      if (?Sc::?C1, ?Ileft:?Sc)
                                      then (false)
                                      else (true)})
  else (?Ileftlst=collectset{?Ileft|?Ileft:?C1}),
  %uvinsertleft(?Ileftlst,?C1,?C2,?Deep),
  if   (?Deep="extent")
  then (?Irightlst=collectset{?Iright|?Iright:?C2,
                                      if (?Sc::?C2, ?Iright:?Sc)
                                      then (false)
                                      else (true)})
  else (?Irightlst=collectset{?Iright|?Iright:?C2}),
  %uvinsertright(?Irightlst,?C1,?C2,?Deep).

  %uvinsertleft([?H|?Ileft],?C1,?C2,?Deep):-
    insert { iuv(?C1,?C2,?Deep,?H, 1):uv(?C1,?C2,?Deep) },
    %uvinsertleft(?Ileft,?C1,?C2,?Deep).
  %uvinsertleft([],?_,?_,?_).
  %uvinsertright([?H|?Iright],?C1,?C2,?Deep):-
    insert { iuv(?C1,?C2,?Deep,?H, 2):uv(?C1,?C2,?Deep) },
    %uvinsertright(?Iright,?C1,?C2,?Deep).
  %uvinsertright([],?_,?_,?_).
```

Procedure **%UVINSERTLEFT** (and **UVINSERTRIGHT**) is recursive and takes four arguments where all of which are instantiated in **%UVDATALOID,** i.e. the calling procedure. The first argument, that drives the recursion, is the list of objects to create an identifier for. On each procedure invocation an object instance-of assertion is inserted into the object base (i.e. **IUV(…):UV(…)**). The instance identifier, **IUV(…)** is a composition of class instances, extent indicator, the head of the list (i.e. first argument), and an arbitrary number. The latter is the only difference between **%UVINSERTLEFT** and **%UVINSERTRIGHT** with values '**1**' and '**2**' respectively. An example logical identifier is:

```
iuv(student,exstudent,"deepextent",mary,1)
```

The procedure **%UVDATA** also has three arguments that are unified in procedure **%UV**; these are the class instances and the extent directive. The main idea of this procedure is to 'move' properties from either argument instances to the result collection. Since there are four properties of interest (two related to inheritance and two to arity) and two arguments (i.e. arbitrary mapped with '**1**' and '**2**') then each of the eight cases are collected and an appropriate recursive procedure is invoked; these are called **%UNISERTPMETHOF** and **%UNISERTSMETHOD** and deal with arity and nonarity methods. The collection predicate for inheritable but non arity methods is unified to variable **?LEFTISMLST** which loops over all

objects that have their fifth parameter set to '**1**' (i.e. an instance of **?C1**) and are in an instance-of the union operation being computed, loops over all inheritable but non arity methods (i.e. **?M*=>?D, NOT COMPOUND(?M)**), and it finally relates actual instances-of **?C1** (i.e. **?I**) property values and is bound to **?R**. The **?WL** structure is used to build a tuple and it is actually this tuple that the **COLLECTSET** aggregates.

```
%uvdata(?C1,?C2,?Deep):-
  ?LeftIsmlst=collectset{ ?Wl |
       iuv(?C1,?C2,?Deep,?I,1)[],
       uv(?C1,?C2,?Deep)[?M*=>?D],
       not compound(?M)@_prolog,
       ?I:?C1[?M->?R],
       dt(?D,?Dt),
       ?R:?Dt,
       ?Wl=f(?C1,?C2,?Deep,?I,1,?M,?R) },
  %uvinsertsmethod(?LeftIsmlst),
  …
  ?LeftIpmlst=collectset{ ?Yl |
       iuv(?C1,?C2,?Deep,?I,1)[],
       uv(?C1,?C2,?Deep)[?M(?Pd)*=>?D],
       ?I:?C1[?M(?Pr)->?R],
       dt(?D,?Dt),
       ?R:?Dt,
       dt(?Pd,?Pdt),
       ?Pr:?Pdt,
       ?Yl=f(?C1,?C2,?Deep,?I,1,?M,?Pr,?R) },
  %uvinsertpmethod(?LeftIpmlst),
  … .

      %uvinsertsmethod([?H|?Rlst]):-
            ?H=f(?C1,?C2,?Deep,?I,?N,?M,?R),
            insert{ iuv(?C1,?C2,?Deep,?I,?N)[?M->?R] },
            %uvinsertsmethod(?Rlst).
      %uvinsertsmethod([]).

      %uvinsertpmethod([?H|?Rlst]):-
            ?H=f(?C1,?C2,?Deep,?I,?N,?M,?Pr,?R),
            insert{ iuv(?C1,?C2,?Deep,?I,?N)[?M(?Pr)->?R] },
            %uvinsertpmethod(?Rlst).
      %uvinsertpmethod([]).
```

The procedures **%UNISERTPMETHOF** and **%UNISERTSMETHOD** are identical except for asserting methods with arity. Each invocation of the procedure is passed a list of tuples – those just created with **COLLECTSET** and built the **?WL** tuple, which has to be iteratively traversed to assert the proper object property: i.e. **IUV(?C1,?C2,?DEEP,?I,?N)[?M->?R]**.

Properties of the union operator

An important property of this union operator here is that it is value based. As a consequence of being value based there is a possibility that the operator is *lossy*; for example processing a self union produces an output which is not equal to the input. (In a

previous example we have seen that a number of person objects, with different identifiers, had common property values and consequently these duplicates had to be eliminated).

Every relational operator has a number of algebraic properties: for example union is commutative, associative, and idempotent. In our algebra the union is commutative, idempotent, and mostly associative. These properties give an algebra a number of equivalence transformation rules that are based on structure. We first describe commutatively and then associativity. We briefly state its idempotent properties.

The structural equivalence of union by commutative state that the order of the operands is non- influential to the result (where Ei is a range – e.g. **CLASS** instance):

$$\mathbf{E_1} \cup \mathbf{E_2} \Leftrightarrow \mathbf{E_2} \cup \mathbf{E_1}$$

Commutativity implies, in our framework, that if the following is computed:

```
?- algebra[%uv(student,exstudent,"extent")].
```

Then it implies the following (and therefore no need to re-compute it):

```
?- algebra[%uv(exstudent,student,"extent")].
```

We propose a solution through F-logic (and Flora-2) use of identity (i.e. **:=:** operator). In Flora-2 manual [YANGG08] it is made amply clear that the implementation of this congruence axiom is limited due to its computational costs; nonetheless what is available is within adequate requirement of this project. Basically we want to assert the fact that the following two logical identifiers represent the same object and therefore each object has the same extent (i.e. result).

```
?- uv(exstudent,student,"extent"):=:uv(student,exstudent,"extent").
Yes
?- ?I:uv(student,exstudent,"extent").
?I = iuv(exstudent,student,"extent",mary,2)
…
?I = iuv(exstudent,student,"extent",william,1)
Yes
?- ?I:uv(exstudent,student,"extent").
?I = iuv(exstudent,student,"extent",mary,2)
…
?I = iuv(exstudent,student,"extent",william,1)
Yes.
```

To assert this equality we need a reified rule and an additional step in the union evaluation. The rule is attached to **UVF.ALGQUERYCOMMU** and states that if there is an **UV(C1,C2,"…")** object which is an instance-of **ALGQUERY** then there is an identical object reference with its classes juxtaposed; i.e. **UV(C2,C1,"…")**.

```
uvf[ algquerycommu(?C1,?C2,?Param1) ->
     ${ (  uv(?C1,?C2,?Param1):=:uv(?C2,?C1,?Param1) :-
                  uv(?C1,?C2,?Param1):algquery[uctext->?_Text]  ) }].
```

The code of the union by value procedure then needs alterations: basically a pre-test and a post-action. The pre-test needs to check, whether the union query is an instance-of **ALGQUERY**; i.e. already computed. While the post-action entails the materialisation of the reified rule in **UVF.ALGQUERYCOMMU** and inserting the rule into the object base.

The structural equivalence of union by associativity state that the order of doing two unions in cascade is non - influential to the result:

$$(\mathbf{E_1} \cup \mathbf{E_2}) \cup \mathbf{E_3} \Leftrightarrow \mathbf{E_1} \cup (\mathbf{E_2} \cup \mathbf{E_3})$$

Within this framework union associativity has limitations. The above identity works if the union compatibility mode is 'self' or 'all properties match' in either union. If the mode is an *ISA* there are issues: the main one is that although each pair can be in an *ISA* relationship, it might not be the case in both expressions. (Specifically, if **E1::E2** and **E3::E2** then **E1** and **E3** are not *ISA* related). An improvement is possible if the union compatibility check confirms implicit *ISA* relationships (and as indicated earlier). Also it is expected that each union has the same extent for the associativity to compute.

The reified rule follows:

```
uvf[ algqueryassoc(?C1,?C2,?Param1) -> ${ (

uv(uv(?C1,?C11,?Param1),?C2,?Param1):=:uv(?C1,uv(?C11,?C2,?Param1),?Param
1):-
  uv(uv(?C1,?C11,?Param1),?C2,?Param1):algquery,
  uv(uv(?C1,?C11,?Param1),?C2,?Param1)[uctext->"UC: Self";uctext->"All
match"]) } ].
```

The following script indicates an example:

```
?- algebra[%uv(person,operson,"extent")].
Yes.
?- algebra[%uv(yperson,uv(person,operson,"extent"),"extent")].
Yes.
?- ?Q:algquery.
?Q = uv(person,operson,"extent")
?Q = uv(yperson,uv(person,operson,"extent"),"extent")
Yes.
?- ?Q:algquery,?I:?Q.
?Q = uv(person,operson,"extent")
?I = iuv(person,operson,"extent",dri,1)
…
?Q = uv(yperson,uv(person,operson,"extent"),"extent")
?I =
iuv(yperson,uv(person,operson,"…"),"extent",iuv(person,operson,"…",p2,1),
2)
Yes
?- uv(?C11,?C3,?Param1):algquery[uctext->"UC: Self";uctext->"UC: all
properties match"],?C11=uv(?C1,?C2,?Param1) .
```

```
?C11 = uv(person,operson,"extent")
?C3 = yperson
?Param1 = "extent"
?C1 = person
?C2 = operson
Yes.
```

The union by value operator is idempotent in the sense that repeated application of union on the same operands does not change the output.

11.2.3.3 Difference

The binary difference algebraic operator meaning for union compatible arguments is such that any object that is an instance-of the first argument but not of the second argument is included in the result. The operator is not commutative. The data-type signature of the output is determined by a data-type signature relationship of the input arguments. Set difference is colloquially known as minus. There are two issues specific to our data and query model: the first is whether the extent or deep extent of the arguments is required; the second being the demands that this operator is value based and produces a set (i.e. requires a value-based comparison and a duplicate check with elimination).

An example of value-based difference is finding value-based instances of **CLASS PERSON** that are not instances of **STUDENT CLASS**; the example is expecting *ISA* inference. The verbose output confirms union compatibility, and query related instantiations proceed very much like what happen in the union operation. The operator has to take a turn for its own because it needs to remove instances; first it removes common instances by value with the second operand range, and secondly any duplicate objects, by value, present in the result's computation. The latter duplication check is identical to the one found in union by value.

```
?- algebra[%mv(person,student,"deepextent")].
union compatible
UC: ok
UC: $2 ISA $1 related
algquery instance created
signatures created
data loid done
data move done
?C1lst […]
?C2lst […]
Diff to purge list: […]
Diff deletion done
duplicate list[…]
duplicate deletion done
Yes.
?- ?Q:algquery,?I:?Q.
?Q = mv(person,student,"deepextent") ?I =
imv(person,student,"deepextent",dri,1)
?Q = mv(person,student,"deepextent") ?I =
imv(person,student,"deepextent",j,1)
```

```
?Q = mv(person,student,"deepextent") ?I =
imv(person,student,"deepextent",p1,1)
?Q = mv(person,student,"deepextent") ?I =
imv(person,student,"deepextent",p2,1)
Yes.
```

The second part of the script shows that difference by value is an instance-of **ALGQUERY** object and is identified by a template of **%MV(ARG1,ARG2,PARMA1)** structure and its instances are identified by **IMV(…)** template. Again this is similar to the union by value except for the functors.

The difference operator uses reified rules too. The first two, **ALGQUERYINSTANCE** and **ALGQUERYDT**, deal with **ALGQUERY** instantiation and data type signature creation. The following code snippet shows that these rules are properties of object **MVF**; i.e. the supporting object for difference operator. (Note there are actually four instantiations of **ALGQUERYDT** and each related to one positive union compatible determination). The methods that are assigned reified rules take arguments and these have an identical meaning to that of the union by value presented in an earlier section.

```
mvf[algqueryinstance(?C1,?C2,?Param1,?Text) ->
    ${ (  mv(?C1,?C2,?Param1):algquery[uctext->?Text]:- true  ) },

    algquerydt(?C1,?C2,?Param1,"UC:  all properties match") ->
    ${ (  mv(?C1,?C2,?Param1)[?M{?B:?T}*=>?D] :-
            ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog),
       (  mv(?C1,?C2,?Param1)[?M{?B:?T}=>?D]  :-
            ?C1[?M{?B:?T}=>?D],  not compound(?M)@_prolog),
       (  mv(?C1,?C2,?Param1)[?M(?P){?B:?T}*=>?D] :-
            ?C1[?M(?P){?B:?T}*=>?D]),
       (  mv(?C1,?C2,?Param1)[?M(?P){?B:?T}=>?D]  :-
            ?C1[?M(?P){?B:?T}=>?D] ) }].
```

The following query shows the data signature of the example difference query just presented.

```
?- ?Q:algquery[?M*=>?Dt;?M=>?Dt].
?Q = mv(person,student,"deepextent") ?M = fname  ?Dt = string
?Q = mv(person,student,"deepextent") ?M = telno  ?Dt = integer
…
Yes.
```

The operation difference by value implementation, **ALGEBRA[%MV(…)]**, is similar to union. First union compatibility is checked and its mode determined; then the query instance and output signature are created through the procedure's arguments and rule reification. The semantics of procedure **%MVDATALOID** is a one sided copy of **%UVDATALOID** as only the left hand side instances, i.e. of **?C1**, need instantiating. Basically the procedure creates an instance-of assertion for the query object in the form of **IMV(…):MV(…)**. Likewise

procedure **%MVDATA** is a one sided copy of **%UVDATA**; the procedure copies property values from instances of the **CLASS ?C1** to instances of **MV(…)**.

```
algebra[%mv(?C1,?C2,?Deep)] :-
  algebra[%unioncomp(?C1,?C2,?Text)],
  writeln('UC: ok')@_prolog,
  writeln(?Text)@_prolog,
  mvf[ algqueryinstance(?C1,?C2,?Deep,?Text) -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  mvf[ algquerydt(?C1,?C2,?Deep,?Text)-> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  %mvdataloid(?C1,?C2,?Deep),
  writeln('data loid done')@_prolog,
  %mvdata(?C1,?C2,?Deep),
  writeln('data move done')@_prolog,
  %mvdatadiffpurge(?C1,?C2,?Deep,[],?Difflst),
  write('Diff to purge list: ')@_prolog,writeln(?Difflst)@_prolog,
  algebra[%delduplicate(?Difflst,mv(?C1,?C2,?Deep))],
  writeln('Diff deletion done')@_prolog,
  algebra[%duplicates(mv(?C1,?C2,?Deep),?Duplst)],
  write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
  algebra[%delduplicate(?Duplst,mv(?C1,?C2,?Deep))],
  writeln('duplicate deletion done')@_prolog.
```

The coding takes a distinct slant in procedure **%MVDATADIFFPURGE** which is responsible for finding and purging instances moved to **MV(…)** but are also in **?C2** – i.e. should not be in the operation's output. The procedure takes five arguments: the first three are the two operands and the *ISA* inference indicator; the last two arguments are the start and end lists of instances-of **MV(…)** that need retracting. The calling procedure, **ALGEBRA[%MV(…)]**, binds the first four arguments. On invocation the procedure creates two lists reflecting instances-of relationships: the first is of **IMV(…):MV(…)** just created–**?C1LST**, and the second is of instances-of **?C2** (i.e. the second operand)–**?C2LIST**. This procedure calls **%MINUSLOID** that takes five arguments: the first is the class instance **?C1**, the next two are the lists just created, and the last two are the start and end list of instances that are to be retracted. Procedure **%MINUSLOID** is recursive and iterates on the list of **?C1** instances; on each of its invocation the it calls another recursive procedure called **%MINUSLOIDINNER** that for the current instance of **?C1** all instances of **?C2** are checked for value duplication. The duplicate check is done to **ALGEBRA[%DUPLICATE(…)]** procedure **%DUPL_CHK_COMP_DIFF** that compares two instances by value. (The procedure **ALGEBRA[%DUPLICATE(…)]** explanation is forthcoming). If there is duplication then we halt comparison with **?C2** instances and push the duplicate object (i.e. an instance of **?C1**) onto the duplicate list. Once a duplicate list is built, in **%MINUSLOID**, it is passed back to

calling **ALGEBRA[%MV(…)]** to continue the final steps. Of course the next step is to retract instances identified in **%MVDATADIFFPURGE**. The last step of the operator is to ensure that their no duplicates in the result set – basically it is possible that there are duplicates in instances of **?C1** too.

```
%mvdatadiffpurge(?C1,?C2,?Deep,?Sduplst,?Duplst) :-
  ?C1lst=collectset{?Ii|imv(?C1,?C2,?Deep,?I,1):mv(?C1,?C2,?Deep),
                       ?Ii=imv(?C1,?C2,?Deep,?I,1)},
  if  ( ?Deep = "extent" )
  then (?C2lst=collectset{?H|?H:?C2, if (?Sc::?C2, ?H:?Sc)
                                     then (false)
                                     else (true)})
  else (?C2lst=collectset{?H|?H:?C2}),
  write('?C1lst ')@_prolog,writeln(?C1lst)@_prolog,
  write('?C2lst ')@_prolog,writeln(?C2lst)@_prolog,
  %minusloid(?C1,?C1lst,?C2lst,?Sduplst,?Duplst).

   %minusloid(?C1,[?I|?C1lst],?C2lst,?Sduplst,?Duplst) :-
     %minusloidinner(?C1,?I,?C2lst,?Result),
     if   (?Result=[])
     then (%minusloid(?C1,?C1lst,?C2lst,?Sduplst,?Duplst))
     else (%minusloid(?C1,?C1lst,?C2lst,[?I|?Sduplst],?Duplst)).
   %minusloid(?_,[],?_,?Duplst,?Duplst):- !.

     %minusloidinner(?C1,?I,[?H|?C2lst],?Result) :-
       if   ( \+ %dupl_chk_comp_diff(?C1,?I,?H) )
       then ( ?Result=[?I] )
       else ( %minusloidinner(?C1,?I,?C2lst,?Result) ).
     %minusloidinner( ?_,?_,[],[] ):- !.
```

Properties of the difference operator

The difference by value is neither commutative nor associative. Nonetheless it is an important operator to express queries. On the other hand it is a non-monotonic operator. At a computational level it is expensive to compute.

11.2.3.4 Duplicate Elimination

The high-level requirements for duplicate identification and purging are: given a **CLASS** instance-of (or **STRUCTURE**, or **ALGQUERY**), do any of its instances have value equivalence; and for any duplicates instances needs to be purged (but leaving one representation).

The procedure **%DUPLICATES**, attached to object **ALGEBRA**, takes two arguments. The first is a **CLASS** instance (i.e. **?C**), and is bound in the calling procedure, and the second argument is a list of objects (**?DUPLST**) that are duplicate and this is its output. The procedure starts by checking that the **CLASS** instance is a user-defined object or an **ALGQUERY** object, it creates a list of all its instances, and then calls a recursive procedure **%DUPL_CHK** that takes four arguments. This procedure iterates on the list of instances, i.e. the second argument, and invokes a procedure **%DUPL_CHK_COMP** to check if any instance-of

the first argument has a value based copy to the head of the list (i.e. **?I**). Each invocation of **%DUPL_CHK_COMP** returns a list of duplicates and this is appended to **%DUPL_CHK** third argument. When recursion for **%DUPL_CHK** ends because list of instances is exhausted then its third and fourth argument are unified.

In procedure **%DUPL_CHK_COMP** it needs to determine if all properties match in value. The pattern to determine this is in double negative: it is not the case that two instances have a different value in a property. It is to be noted that **?I** is compare not to all instances of the class but what is left of the list; note how the head and rest of list in **%DUPL_CHK** second argument are passed separately in **%DUPL_CHK_COMP** second and third arguments. To check for this **%DUPL_CHK_COMP** calls procedure **%DUPL_CHK_COMP_DIFF**. The latter has four versions to take care of the four different data type signatures of interest.

```
algebra[%duplicates(?C,?Duplst)] :-
    (?C:class;?C:structure;?C:algquery),
    ?Ilst=collectset{?I|?I:?C},
    %dupl_chk(?C,?Ilst,[],?Duplst).

    %dupl_chk(?C,[?I|?Rlst],?Blst,?Flst):-
      %dupl_chk_comp(?C,?I,?Rlst,[],?Dlsti),
      ?Blst[_append(?Dlsti)->?Ilst]@_basetype,
      %dupl_chk(?C,?Rlst,?Ilst,?Flst).
    %dupl_chk(?_,[],?Flst,?Flst):- !.

      %dupl_chk_comp(?C,?I,[?H|?Rlst],?Blst,?Dlsti):-
        if   ( \+ %dupl_chk_comp_diff(?C,?I,?H) )
        then ( ?Blst[_append([?H])->?Ilst]@_basetype,
               %dupl_chk_comp(?C,?I,?Rlst,?Ilst,?Dlsti))
        else ( %dupl_chk_comp(?C,?I,?Rlst,?Blst,?Dlsti)).
      %dupl_chk_comp(?_,?_,[],?Dlsti,?Dlsti):- !.

        %dupl_chk_comp_diff(?C,?I,?H) :-
          ?C[?M*=>?D], not compound(?M)@_p, ?I[?M->?R1], ?H[?M->?R2],
          ?R1 != ?R2, (not ?I[?M->?R2] ; not ?H[?M->?R1]).
        %dupl_chk_comp_diff(?C,?I,?H) :-
          ?C[?M=>?D],not compound(?M)@_prolog, ?I[?M->?R1], ?H[?M->?R2],
          ?R1 != ?R2,
          (not ?I[?M->?R2] ; not ?H[?M->?R1]),
          not(?R1=?R2).
        %dupl_chk_comp_diff(?C,?I,?H) :-
          ?C[?M(?P)*=>?D],?I[?M(?P1)->?R1],?H[?M(?P2)->?R2],
          ( ?R1 != ?R2 ; ?P1 != ?P2 ),
          (not ?I[?M(?P2)->?R2]; not ?H[?M(?P1)->?R1] ).
        %dupl_chk_comp_diff(?C,?I,?H) :-
          ?C[?M(?P)=>?D],?I[?M(?P1)->?R1],?H[?M(?P2)->?R2],
          ( ?R1 != ?R2 ; ?P1 != ?P2 ),
          (not ?I[?M(?P2)->?R2]; not ?H[?M(?P1)->?R1] ).
```

This procedure is computationally expensive even though coding tries to minimise comparisons. In later sections techniques are shown on how decrease its computation cost or avoid its evaluation.

The purging of duplicates once identified is straight forward. The procedure **%DELDUPLICATE** is attached to object algebra and takes two arguments. The first argument is a list of objects to delete and the second argument is the **CLASS** instance. The procedure calls a recursive procedure **%PROC_DELDUPLICATE** to iterate on the list of objects and for each invocation deletes the head of the list by retracting the instance-of assertion (i.e. **?H:?C**). The retraction is done through Flora-2 object base operation **DELETEALL{…}**.

```
algebra[%delduplicate(?Duplst,?C)] :- %proc_delduplicate(?Duplst,?C).

  %proc_delduplicate([?H|?Duplst],?C) :-
    deleteall{?H:?C},
    %proc_delduplicate(?Duplst,?C).
  %proc_delduplicate([],?_).
```

11.2.3.5 Intersection

The binary intersection algebraic operator meaning for union compatible classes insists that any object included in the result must have an instance-of in both arguments. Also the data type signature of the output is determined by a data type signature relationship of its input arguments. There are two issues specific to our data and query model: the first is whether the extent or deep extent of the arguments is required; the second being the demands that this operator is value based and produces a set (i.e. requires duplicate check and elimination).

Also it is well known that intersection is derived from difference and union, or difference operators. The following identity holds (where Ei is for example a **CLASS** instance):

$$\mathbf{E_1} \cap \mathbf{E_2} \Leftrightarrow (\mathbf{E_1} - (\mathbf{E_1} - \mathbf{E_2}))$$

Intersection, like union by value, is commutative and associative. Also, as discussed for the union operator, interaction by value is lossy.

In our framework this works as follows: objects **STUDENT** and **POSTGRAD** are instances-of **CLASS** and furthermore **POSTGRAD** is *ISA* related to **STUDENT**. The intersection of their deep extent is equal to **POSTGRAD'S** deep extent. The following script shows that the intersection of these deep extents produces does produce postgrad deep extent (i.e. derived from object **SUSAN**). Note the output's data signature and its properties assignment.

```
?- algebra[%mv(student,postgrad,"deepextent")].
Yes.
?- ?Q:algquery.
?Q = mv(exstudent,student,"deepextent")
…
Yes.

?- ?I:mv(student,postgrad,"deepextent").
?I = imv(student,postgrad,"deepextent",mary,1)
…
?I = imv(student,postgrad,"deepextent",robert,1)
Yes.
?- algebra[%mv(student,mv(student,postgrad,"deepextent"),"deepextent")].
?- ?I:mv(student,mv(student,postgrad,"deepextent"),"deepextent").
?I = imv(student,mv(student,postgrad,"deepextent"),"deepextent",susan,1)
Yes.
?- ?I:postgrad.
?I = susan
Yes.
?-
mv(student,mv(student,postgrad,"deepextent"),"deepextent")[?M*=>?Dt;?M=>?
Dt].
?M = enrolon          ?Dt = course
?M = fname            ?Dt = string
?M = stage            ?Dt = string
?M = telno            ?Dt = integer
?M = result(string)   ?Dt = unit
Yes.
?- ?I:mv(student,mv(student,postgrad,"deepextent"),"deepextent")[?M->?R].
?I = imv(…)    ?M = enrolon   ?R = bsc_eng
?I = imv(…)    ?M = fname     ?R = "susan"^^_string
?I = imv(…)    ?M = stage     ?R = "s"^^_string
…
Yes.
```

11.2.4 Project

The unary project by value operator produces a set of objects from a range whose properties are the ones passed on to the operator. The operand takes an indicator as to whether deep extent or extent of a range is required. Other than ignoring non-listed properties from the target class instance the operator has to ensure the output is devoid of duplicates by value. The project operator is known to be a constructor operator in database algebras as it builds a structure from the input structure; albeit with fewer properties. Also the data-type signature of the output is determined by a data-type signature relationship of input argument and the list of properties to project.

A basic example that projects by value on two attributes follows: project properties **FNAME** and **TELNO** from **CLASS** instance **PERSON'S** deep extent. The invocation of operator called **ALGEBRA[%PV(…)]** shows it takes three arguments: the class range, *ISA* indicator, and the list of properties to project. The script shows that no projected properties are undefined for **CLASS PERSON**, and finally the duplicate test has passed. The query instance and query's

instances-of are also shown. In the last query the projected values are shown (with output heavily sanitised).

```
?- algebra[%pv(person,"deepextent",[fname,telno])].
method not found list: []
Methods exist: ok
algquery instance created
signatures created
data loid and move done
duplicate list[…]
duplicate deletion done

Yes.
?- ?Q:algquery,?I:?Q.
?Q=pv(person,"deepextent",_#'17253)
?I=ipv(person,"deepextent",_#'17253,dri)
?Q=pv(person,"deepextent",_#'17253)..?I=ipv(person,"deepextent",_#'17253,
j)
…
?Q=pv(person,"deepextent",_#'17253)..?I=ipv(person,"deepextent",_#'17253,
susan)
Yes.
?- ?_Q:algquery,?I:?_Q[?M->?Rt].
?I = ipv(person,"deepextent",_#'17253,dri)  ?M = fname  ?Rt =
"dri"^^_string
?I = ipv(person,"deepextent",_#'17253,dri)  ?M = telno  ?Rt =
"238751"^^_integer
?I = ipv(person,"deepextent",_#'17253,j)    ?M = fname  ?Rt =
"joe"^^_string
?I = ipv(person,"deepextent",_#'17253,j)    ?M = telno  ?Rt =
"224502"^^_integer
…
Yes.
```

The following example shows project by value on properties that include a method with an argument: project out the **FNAME** and **RESULT(…)** properties for all instances-of **STUDENT** (but not its deep extent). The three queries in the following script depict: first the project by value operation, second the data signature of the result instance-of query, and the third being the output of the query (i.e. sanitised).

```
?- algebra[%pv(student,"extent",[fname,result(string)])].
method not found list: []
Methods exist: ok
…
Yes.
?- ?Q:algquery[?M*=>?Rt;?M=>?Rt].
?Q = pv(student,"extent",_#'17254)   ?M = fname   ?Rt = string
?Q = pv(student,"extent",_#'17254)   ?M = result(string)  ?Rt = unit
…
Yes
?- ?Q:algquery,?I:?Q[?M->?Rt].
?Q = pv(…)   ?I = ipv(…)   ?M = fname                  ?Rt =
"mary"^^_string
?Q = pv(…)   ?I = ipv(…)   ?M = result("a"^^_string)  ?Rt = idb
?Q = pv(…)   ?I = ipv(…)   ?M = result("b"^^_string)  ?Rt = javaprog
?Q = pv(…)   ?I = ipv(…)   ?M = result("b"^^_string)  ?Rt = pdb
?Q = pv(…)   ?I = ipv(…)   ?M = result("c"^^_string)  ?Rt = dsa
```

The project by value also uses two reified rules; these are attached to an *ad hoc* object with identifier **PVF**. In property **ALGQUERYINSTANCE** the reified rule creates the **ALGQUERY**

instance in the form of **PV(…):ALGQUERY**. The method actually takes four arguments: the first being the class instance, the second is the *ISA* indicator, the third argument is the list of properties to project, and the forth is a unique identifier (to be explained later). Note that the projected list is added to property **PROJECTION** of object **PV(…)** just created.

```
pvf  [algqueryinstance(?C1,?Param1,?Mthlst,?Tag) ->
  ${ (  pv(?C1,?Param1,?Tag):algquery [projection->?Mthlst] :- true )}].
```

The second reification concerns the data type signature of the result and it is attached to property **ALGQUERYDT**. The data type of a projection is determined by the method list to project. The invocation requires four arguments: the first is the **CLASS** range, second is the *ISA* indicator, the third is a property list, and the forth is a unique identifier. Once an instantiation of a reified rule is requested it builds a rule that for each property, in the list, it copies its data type signature from **?C1** instance to a **PV(…)** instance. The rule uses a procedure called **%LMEMBER** to recursively iterate the method list. Actually four rules are created for four possible data signatures of interest here (i.e. **?M*=>?DT, ?M(?A)*=>?DT, ?M=>?DT, AND ?M(?A)=>?DT**).

```
  pvf  [algquerydt(?C1,?Param1,?Mthlst,?Tag)  ->
        ${ (pv(?C1,?Param1,?Tag)[?M{?B:?T}*=>?D]
              :-  pv(?C1,?Param1,?Tag):algquery,
                  %lmember(?M,?Mthlst),
                  ?C1[?M{?B:?T}*=>?D],
                  not compound(?M)@_prolog),
        …  }].
```

The actual evaluation of project by value is done through procedure **%PV** in object **ALGEBRA**. The procedure **%PV** takes three arguments: the first is a class instance from which to project (i.e. unified to variable **?C1**), the second is an indicator on the extent required (i.e. unified to variable **?DEEP**), and the third argument being the list of properties to project. The first task is to ascertain that the methods in the list properties are indeed found in the class instance data type signature. An *ad hoc* **ALGEBRA** object method is responsible for this and it is called **ALGEBRA[%METHODSEXIST(…)]** – see section 11.2.3.1 for its explanation. To use this method a call is made with a **CLASS** instance, i.e. **?C1**, and method list, i.e. **?USMTHLAST**, and if all methods have a data type signature in **CLASS** instance, then an empty list is returned in the third argument; otherwise the return list would have the offending methods without a data type signature. The next actions are to sort the method list and use Flora-2 meta-level construct to generate a unique identifier; it is called *newoid*

– these are required together with the operator invocation in the two reification rules **ALGQUERYINSTANCE** and **ALGQUERDT** of object **PVF**. These rules are instantiated are inserted into the object base. The next action is to create the query object instance-of assertions and this is done through calling procedure **%PVDATALOID(…)**. The assertions are of the form **IPV(…):PV(…)**. Once an identifier is created for an instance then the next step is to assigned values to the projected properties – the procedure **%PVDATA(…)** takes care of this and is explained soon. The last step require that any value duplicates in the result set are identified and retracted: as usual these are taken care by object algebra methods %**DUPLICATES(…)** and **%DELDUPLICATES(…)**.

```
algebra[%pv(?C1,?Deep,?Usmthlst)] :-
    algebra[%methodsexist (?C1,?Usmthlst,?Nomthlst)],
    if   ( ?Nomthlst = [] )
    then ( writeln('Methods exist: ok')@_prolog )
    else ( write('Methods list: problem; missing attributes - ')@_p,
           writeln(?Nomthlst)@_prolog,
           fail),
    call(sort(?Usmthlst,?Mthlst)@_prolog),
    newoid{?Tag},
    pvf[ algqueryinstance(?C1,?Deep,?Mthlst,?Tag) -> ?Rules1 ],
    insertrule { ?Rules1 },
    writeln('algquery instance created')@_prolog,
    pvf[ algquerydt(?C1,?Deep,?Mthlst,?Tag)-> ?Rules2 ],
    insertrule { ?Rules2 },
    writeln('signatures created')@_prolog,
    %pvdataloid(?C1,?Deep,?Mthlst,?Tag),
    %pvdata(?C1,?Deep,?Mthlst,?Tag),
    writeln('data loid and move done')@_prolog,
    algebra[%duplicates(pv(?C1,?Deep,?Tag),?Duplst)],
    write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
    algebra[%delduplicate(?Duplst,pv(?C1,?Deep,?Tag))],
    writeln('duplicate deletion done')@_prolog.
```

The procedure **%PVDATA** is called with its four arguments unified by the calling procedure. In **%PVDATA** a list of tuples is created for each of the four types of data-type signature considered in the framework and related to the arguments passed to the procedure. The aggregate **COLLECTSET** predicate uses **%LMEMBER** predicate to ensure that only methods in the projection list are of interest. Once the list of tuples is created, **?WL**, and aggregated then this list is passed to recursive procedure **%PVINSERTSMETHOD**(…) to assert arity less methods values.

```
%pvdata(?C1,?Deep,?Mthlst,?Tag) :-
  ?LeftIsmlst=collectset{
     ?Wl | pv(?C1,?Deep,?Tag):algquery, %lmember(?M,?Mthlst),
           ipv(?C1,?Deep,?Tag,?I)[],  pv(?C1,?Deep,?Tag)[?M*=>?D],
           not compound(?M)@_prolog,
           ?I:?C1[?M->?R],dt(?D,?Dt),?R:?Dt,
           ?Wl=f(?C1,?Deep,?Tag,?I,?M,?R) },
  %pvinsertsmethod(?LeftIsmlst),
  …  .
```

```
        %pvinsertsmethod([?H|?Rlst]):-
          ?H=f(?C1,?Deep,?Tag,?I,?M,?R),
          insert{ ipv(?C1,?Deep,?Tag,?I)[?M->?R] },
          %pvinsertsmethod(?Rlst).
        %pvinsertsmethod([]).
```

Properties of the project operator

One of the few relevant properties is that projection by value is idempotent. Another property is that successive projects on the same class are equal to the last project; of course each successive projected method list must be a subset of the previous: (Pi in caps is the customary symbol for projection and class denotes ranges – e.g. **CLASS** instance).

$$\prod_{A_n}\left(\prod_{A_{n-1}}\cdots\left(\prod_{A_1}(class)\right)\cdots\right)=\prod_{A_n}(class)$$

$$where\ A_n \subseteq A_{n-1} \subseteq \cdots \subseteq A_1.$$

11.2.4.1 Check existence of methods in a data type signature

The procedure to check if a methods list exists is found in object **ALGEBRA** and is called **%METHODSEXIST**. The procedure takes three arguments the first two are the class instance and the method list to check while the third is a list of methods that have not been found in the current class instance. While the first two arguments are unified by the calling procedure the third is return my procedure **%METHODSEXIST**. To iterate over the methods in the list the procedure calls a recursive procedure **%PROC_METHODSEXIST** with the first three arguments unified and a forth for the return result. The first two correspond to the class instance and the methods list to check. A simple conjunctive test checks if data type signature for method exists; if not it is appended to the not found list. Once all methods are tested the recursion terminates and result is unified and passed on to calling procedures.

```
algebra[%methodsexist(?C1,?Mthlst,?Nomthlst)]:-
            %proc_methodsexist(?C1,?Mthlst,[],?Nomthlst),
            write('method not found list: ')@_prolog,
            writeln(?Nomthlst)@_prolog.

    %proc_methodsexist(?C1,[?M|?Mthlst],?S,?Nomthlst):-
        if (?C1[?M*=>?_];?C1[?M=>?_])
        then (%proc_methodsexist(?C1,?Mthlst,?S,     ?Nomthlst))
        else (%proc_methodsexist(?C1,?Mthlst,[?M|?S],?Nomthlst)).
    %proc_methodsexist(?C1,[],?Nomthlst,?Nomthlst) :- !.
```

The following script shows an invocation of the procedure with an arbitrary list of methods that are checked for presence in **CLASS** instance **STUDENT**. The script shows that method **RESULT** is not present but **RESULT(STRING)** is.

```
?- ?C=student, ?C[?M*=>?Dt; ?M=>?Dt].
?C = student   ?M = fname            ?Dt = string
…
?C = student   ?M = result(string)   ?Dt = unit
Yes.
?- ?C=student, ?Mthlst=[fname, result, result(string)],
   algebra[%methodsexist(?C,?Mthlst,?Nogood)].
method not found list: [result]
?C = student   ?Mthlst = [fname, result, result(string)]  ?Nogood =
[result]
Yes.
```

11.2.5 Product

Product by value is based on the set theoretic operand (i.e. Cartesian product), it is a binary operation between two classes, and it is used to build other structures. For product by value to work there must be no property name in common in its two input classes. The result's structure of a product by value is a concatenation of both arguments' respective structures. The result's content is the pairing of every instance of one argument with every instance of the second argument. There are two issues specific to our data and query model: the first is whether the extent or deep extent of the arguments is required; the second being the demands that this operator is value based and produces a set (i.e. requires duplicate check and elimination). Another particular issue is self product, which is a basis of self join, and its evaluation. As explained this is not allowed unless one operand has its properties renamed to avoid naming clash in the result.

In the following example a product by value is executed for **CLASS** instances **UNIT** and **STUDENT**. The operator is asked to consider the deep extent. The verbose output confirms no property, by name and data type, is found in both **CLASS** instances, and indicates that no duplicates by value are present either. The query's meta instance (i.e. instance-of **ALGQUERY**) and query result are shown in the script below (e.g. instances-of **XV(UNIT,STUDENT,"DEEPEXTENT")**. In the middle part of the script the data type signature of the result is indicated. Also the aggregate queries at the end of the script, and given there are no duplicates, confirm the output size, i.e. the cardinality of the output, is equal to the multiplication of the cardinality of the arguments.

```
?- algebra[%xv(unit,student,"deepextent")].
Nameclash: none / ok
algquery instance created
signatures created
data loid and move done
duplicate list[]
duplicate deletion done
Yes.
?- ?Q:algquery,?Q[?M*=>?Dt;?M=>?Dt].
```

```
?Q = xv(unit,student,"deepextent") ?M = applicableto ?Dt = course
?Q = xv(unit,student,"deepextent") ?M = assessment   ?Dt = string
?Q = xv(unit,student,"deepextent") ?M = coordby      ?Dt = person
?Q = xv(unit,student,"deepextent") ?M = credits      ?Dt = integer
?Q = xv(unit,student,"deepextent") ?M = enrolon      ?Dt = course
?Q = xv(unit,student,"deepextent") ?M = fname        ?Dt = string
…
Yes.
?- ?_Q:algquery,?I:?_Q, ?I[uname->?U,fname->?N].
?I = ixv(…)   ?U = "group work"^^_string     ?N = "mary"^^_string
?I = ixv(…)   ?U = "group work"^^_string     ?N = "michael"^^_string
?I = ixv(…)   ?U = "group work"^^_string     ?N = "nancy"^^_string
…
?I = ixv(…)   ?U = "dist db"^^_string        ?N = "nancy"^^_string
…
72 solution(s) in 0.0000 seconds
Yes.
?- ?Q:algquery,?Icnt=count{?I|?I:?Q}.
?Q = xv(unit,student,"deepextent")  ?Icnt = 72   ?I = ?_h3814
Yes.
?- ?C=student,?Ic=count{?I|?I:?C}.
?C = student                        ?Ic = 8      ?I = ?_h3687
Yes.
?- ?C=unit,?Ic=count{?I|?I:?C}.
?C = unit                           ?Ic = 9      ?I = ?_h3675
Yes.
```

The following script is based around an example of a self product on **CLASS** instance **COURSE**. The verbose output notes that signatures created for the result are cognisant of self product and duplication check is executed but without any occurrence. The first query lists the instances, while the next two compare the data-type signature of **COURSE** and its self product (at least for inheritable methods). In the data-type signature of the product result note that the method names have been renamed to avoid any name conflict. The last query shows a part of the output.

```
?- algebra[%xv(course,course,"deepextent")].
Nameclash: none / ok
algquery instance created
signatures created (for self product)
data loid and move done
duplicate list[]
duplicate deletion done
Yes.
?- ?Q:algquery,?I:?Q.
?Q = xv(…) ?I = ixv(course,course,"deepextent",ba_english,ba_english)
?Q = xv(…) ?I = ixv(course,course,"deepextent",ba_english,bsc_comp)
?Q = xv(…) ?I = ixv(course,course,"deepextent",ba_english,bsc_eng)
…
Yes.
?- course[?M*=>?Rt].
?M = cdesc     ?Rt = string
?M = cname     ?Rt = string
Yes.
?- ?Q:algquery,?Q[?M*=>?Dt].
?Q = xv(course,course,"deepextent")  ?M = lf_cdesc  ?Dt = string
?Q = xv(course,course,"deepextent")  ?M = lf_cname  ?Dt = string
?Q = xv(course,course,"deepextent")  ?M = rt_cdesc  ?Dt = string
?Q = xv(course,course,"deepextent")  ?M = rt_cname  ?Dt = string
Yes.
```

```
?- ?_Q:algquery,?I:?_Q[?M->?R].
?I = ixv(course,course,"deepextent",ba_english,ba_english)
?M = lf_cdesc   ?R = "desc english"^^_string

?I = ixv(course,course,"deepextent",ba_english,ba_english)
?M = lf_cname   ?R = "english"^^_string

?I = ixv(course,course,"deepextent",ba_english,bsc_comp)
?M = lf_cdesc   ?R = "desc english"^^_string

?I = ixv(course,course,"deepextent",ba_english,bsc_comp)
?M = lf_cname   ?R = "english"^^_string
…
Yes.
```

For the product operator implementation two types of reified rules are needed. These are attached to an object identified with **XVF**.

The property **ALGQUERYINSTANCE** has a parameterised version and thus allows an assertion that object **XV(…)** is an instance-of **ALGQUERY**. The method takes three arguments: the first two are the collections on which the product is to be executed, and the third contains an indicator to state whether a deep extent or an extent is required for the result.

```
xvf  [algqueryinstance(?C1,?C2,?Deep) ->
  ${ (  xv(?C1,?C2,?Deep):algquery :- true  ) } ].
```

The second reification concerns the data type signature of the result. As stated earlier the result's data type signature is a concatenation of the two classes' own data-type signatures. For clarity let's call these classes the left and right arguments. We have two cases: the first when the left and right are distinct in data type signatures; and the second when they are identical as in self product (at the logical-identifier level). In the implementation here we have opted to have two sets of rules that correspond to the cases just noted.

For the first case we need a set of rules to copy from the right data type signature, and another set to copy from the left. We have previously indicated that we are interested in four types of signature in each class declaration; consequently we need eight rules to cover the product by value result data type signatures. The method **ALGQUERYDT** takes three arguments, that are instantiated by a calling procedure, and these are the left class, right class, and *ISA* indicator. For each argument **CLASS** instance and method signature of interest the rules copy the method signature from a class instance and asserts it for the result's data type. The following, a fragment of the whole, depicts the reified rules that

cater for the left class. For example all inheritable methods without arity of left class, i.e. **?C1**, are included in the product's objects; i.e. **XV(…)**.

```
xvf[
  algquerydt(?C1,?C2,?Deep) ->
  ${   (xv(?C1,?C2,?Deep)[?M{?B:?T}*=>?D]
          :- ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog),
       (xv(?C1,?C2,?Deep)[?M{?B:?T}=>?D]
          :- ?C1[?M{?B:?T}=>?D],  not compound(?M)@_prolog),
       (xv(?C1,?C2,?Deep)[?M(?P){?B:?T}*=>?D]
          :- ?C1[?M(?P){?B:?T}*=>?D]),
       (xv(?C1,?C2,?Deep)[?M(?P){?B:?T}=>?D]
          :- ?C1[?M(?P){?B:?T}=>?D] ),
       … ].
```

The reified rules for the second case are similar to the first except we need to rename the properties to avoid a name clash; in fact there are eight rules to instantiate. We have implemented the following renaming action through procedure **ADDPREFIX(…)**: properties of the left class have '**LF_**' prefixed to its name, and properties of the right class have '**RT_**' prefixed to its name.

```
xvf[
  algquerydtself(?C1,?C2,?Deep) ->
  ${   (xv(?C1,?C2,?Deep)[?Mp{?B:?T}*=>?D]
    :- ?C1[?M{?B:?T}*=>?D], not compound(?M)@_p,
       addprefix('lf_',?M,?Mp)),
       (xv(?C1,?C2,?Deep)[?Mp{?B:?T}*=>?D]
    :- ?C2[?M{?B:?T}*=>?D], not compound(?M)@_p,
       addprefix('rt_',?M,?Mp)),
       … ].
```

The actual evaluation of product by value is done through procedure **%XV** in object **ALGEBRA**. The procedure **%XV** takes three arguments: the first two are the **CLASSES'** instances to operate on (i.e. unified to variables **?C1** and **?C2**), and the third is an indicator on the extent required (i.e. unified to variable **?DEEP**). The first task is to determine if the input operands have distinct property names by calling procedure **%NAMECLASH** of object **ALGEBRA**. If a property name clash exists then the operand fails; for clarity a procedure **%NAMECLASHLIST** is called to get the offending properties in a list and output just before failing. (**%NAMECLASH** and **%NAMECLASLIST** of algebra are explained in section 11.2.4.1). The implementation continues by calling, passing arguments, and instantiating the two reified rules in object **XVF** for query instance (i.e. **ALGQUERYINSTANCE**) and query instance data type signature (i.e. **ALGQUERYDT**). The next job is to create the extent of the current query instance (i.e. **IXV(…):XV(…)** instance-of assertions), and build each instance-of properties values. As in the previous operators implementation two appropriate

procedures, called **%XVDATALOID** and **%XVDATA**, are invoked. Once the query's instance values are instantiated then a duplicate by value check on the result is done and if need be duplicates are eliminated.

```
algebra[%xv(?C1,?C2,?Deep)] :-
  if   ( ?C1!=?C2, algebra[%nameclash(?C1,?C2)] )
  then ( algebra[%nameclashlist(?C1,?C2,?Lc1c2smdmd)],
         write('Nameclash: problem - ')@_prolog,
         writeln(?Lc1c2smdmd)@_prolog,
         fail )
  else ( writeln('Nameclash: none / ok')@_prolog ),
  xvf[ algqueryinstance(?C1,?C2,?Deep) -> ?Rules1 ],
       insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  if   ( ?C1!=?C2 )
  then ( xvf[ algquerydt(?C1,?C2,?Deep)-> ?Rules2 ],
         insertrule { ?Rules2 },
         writeln('signatures created')@_prolog)
  else ( xvf[ algquerydtself(?C1,?C2,?Deep)-> ?Rules22 ],
         insertrule { ?Rules22 },
         writeln('signatures created (for self product)')@_prolog),
  writeln('signatures created')@_prolog,
  %xvdataloid(?C1,?C2,?Deep),
  %xvdata(?C1,?C2,?Deep),
  writeln('data loid and move done')@_prolog,
  algebra[%duplicates(xv(?C1,?C2,?Deep),?Duplst)],
  write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
  algebra[%delduplicate(?Duplst,xv(?C1,?C2,?Deep))],
  writeln('duplicate deletion done')@_prolog.
```

The procedure **%XVDATALOID**, listed below, takes three arguments which are unified in the calling procedure **%XV** of the object algebra. Basically the procedure needs to create an identifier for each instance-of related to the procedure's first of two arguments. For this the **COLLECTSET** predicate is used. Because the operator has control on what extent to include in the output, i.e. **?DEEP**, this is reflected in the qualifier inside each **COLLECTSET** invocation. Once the required **COLLECTSET** is bound to each **CLASS** instance then the two sets are passed on to procedure **%XVINSERT** to create the **IXV(…)** identifiers and instance-of assertion. The procedure is recursive on the list of instances of the left class instance list and once invoked calls yet another recursive procedure, called **%XVINSERTINNER**, to iterate over the instances-of found in the right list. In it asserts are made, through an **INSERT{…}** directive, of the relative fact. Once the left class argument is done a similar computation is done for the right class instances.

```
%xvdataloid(?C1,?C2,?Deep) :-
  if   ( ?Deep = "extent" )
  then ( ?Ileftlst=collectset{?Ileft|?Ileft:?C1,
                              if (?Sc::?C1, ?Ileft:?Sc)
                              then (false)
                              else (true)})
  else ( ?Ileftlst=collectset{?Ileft|?Ileft:?C1}),
```

```
    if   (?Deep="extent")
    then (?Irightlst=collectset{?Iright|?Iright:?C2,
                                if (?Sc::?C2, ?Iright:?Sc)
                                then (false)
                                else (true)})
    else (?Irightlst=collectset{?Iright|?Iright:?C2}),
    %xvinsert(?Ileftlst, ?Irightlst, ?C1,?C2,?Deep).

   %xvinsert([?I|?Ileftlst],?Irightlst,?C1,?C2,?Deep) :-
     %xvinsertinner(?I,?Irightlst,?C1,?C2,?Deep),
     %xvinsert(?Ileftlst,?Irightlst,?C1,?C2,?Deep).
   %xvinsert([],?_,?_,?_,?_).

     %xvinsertinner(?I,[?H|?Irightlst],?C1,?C2,?Deep) :-
        insert { ixv(?C1,?C2,?Deep,?I,?H):xv(?C1,?C2,?Deep) },
        %xvinsertinner(?I,?Irightlst,?C1,?C2,?Deep).
     %xvinsertinner(?_,[],?_,?_,?_).
```

The procedure **%XVDATANORM**, parts of which are listed here under, is long but straightforward. (Also there are two versions that correspond to self product and other products). In sequence it creates eight lists of **IXV(…)** instances. Each list is relative to the two classes and four types of methods of interest. In each list a collection of tuples are created, e.g. **?WL** in **?LEFTISMLST**, that contains details to copy from source to target. Each collection's is composed from relative data type signatures and also cater for name overloading as a method data type signature and data expression must match. Once a list is created it is passed on to procedure **%XVINSERTSMETHOD** for it to iteratively insert each method property value through an **INSERT{…}** directive. There is another method called **%XVINSERTPMETHOD** that caters for copy values of methods with a parameter.

```
  %xvdata(?C1,?C2,?Deep):-
    ?LeftIsmlst=collectset{ ?Wl | ixv(?C1,?C2,?Deep,?I,?H)[],
                                  xv(?C1,?C2,?Deep)[?M*=>?D],
                                  not compound(?M)@_prolog,
                                  ?I:?C1[?M->?R], dt(?D,?Dt), ?R:?Dt,
                                  ?Wl=f(?C1,?C2,?Deep,?I,?H,?M,?R) },
    %xvinsertsmethod(?LeftIsmlst),
    … ,
    ?LeftIpmlst=collectset{ ?Yl | ixv(?C1,?C2,?Deep,?I,?H)[],
                                  xv(?C1,?C2,?Deep)[?M(?Pd)*=>?D],
                                  ?I:?C1[?M(?Pr)->?R], dt(?D,?Dt),
                                  ?R:?Dt,dt(?Pd,?Pdt),?Pr:?Pdt,
                                  ?Yl=f(?C1,?C2,?Deep,?I,?H,?M,?Pr,?R) },
    %xvinsertpmethod(?LeftIpmlst),
    … .

    %xvinsertsmethod([?H|?Rlst]):-
      ?H=f(?C1,?C2,?Deep,?I,?N,?M,?R),
      insert{ ixv(?C1,?C2,?Deep,?I,?N)[?M->?R] },
      %xvinsertsmethod(?Rlst).
    %xvinsertsmethod([]).

    %xvinsertpmethod([?H|?Rlst]):-
      ?H=f(?C1,?C2,?Deep,?I,?N,?M,?Pr,?R),
      insert{ ixv(?C1,?C2,?Deep,?I,?N)[?M(?Pr)->?R] },
      %xvinsertpmethod(?Rlst).
    %xvinsertpmethod([]).
```

Properties of the product operator

An important property of this product operator here is that it is value based. Algebraic properties of product on its own are its commutativity and associativity.

From commutativity of product we have that the order of operands does not affect the result:

$$\mathbf{E_1} \times \mathbf{E_2} \Leftrightarrow \mathbf{E_2} \times \mathbf{E_1}$$

Therefore if the following query is computed:

```
?- algebra[%xv(unit,student,"deepextent")].
```

Then it is result equivalent to:

```
?- algebra[%xv(student,unit,"deepextent")].
```

The technique based on equality of query instance used in the union by value section (i.e. 11.2.2.2) is applicable here too. To assert this equality we need a reified rule and an additional step in the product evaluation. The rule is attached to **XVF.ALGQUERYCOMMU** and states that if there is an **XV(C1,C2,…)** object which is an instance-of **ALGQUERY** then there is an identical object reference with its classes juxtaposed; i.e. **UV(C2,C1,…)**.

```
xvf[ algquerycommu(?C1,?C2,?Param1) ->
     ${ (  xv(?C1,?C2,?Param1):=:xv(?C2,?C1,?Param1) :-
                 xv(?C1,?C2,?Param1):algquery  ) }].
```

The code of the product by value procedure then needs alterations: basically a pre-test and a post-action. The pre-test needs to check if the product query is an instance-of **ALGQUERY**; i.e. already computed. While the post-action entails the materialisation of the reified rule in **XVF.ALGQUERYCOMMU** and inserting the rule into the object base.

The structural equivalence of product by associativity states that the order of doing two products in cascade is not significant to the result:

$$(\mathbf{E_1} \times \mathbf{E_2}) \times \mathbf{E_3} \Leftrightarrow \mathbf{E_1} \times (\mathbf{E_2} \times \mathbf{E_3})$$

The reified rule for implementing the product by value associativity follows:

```
xvf[ algqueryassoc(?C1,?C2,?Param1) -> ${ (

xv(xv(?C1,?C11,?Param1),?C2,?Param1):=:xv(?C1,xv(?C11,?C2,?Param1),?Param1):-
  xv(xv(?C1,?C11,?Param1),?C2,?Param1):algquery,
  xv(xv(?C1,?C11,?Param1),?C2,?Param1)) } ].
```

11.2.5.1 Property Name clash

It is sometimes necessary to have two **CLASSES'** set of properties distinct at their name level. Specifically we have operations that presume that two **CLASS** instances, for example,

do not have a property in common by name. Procedure **%NAMECLASH** of object **ALGEBRA**, that takes two arguments, succeeds if indeed such a commonly named property exists within each argument. The procedure calls another procedure called **%NC** that succeeds if a data-type signature coincides in each argument structure. There are four **%NC** rules, each case dealing with a mode of data-type signature. It is important to note that name overlap tolerates data type signature name overloading; i.e. for a method to name clash it must have same name and return data type. The code of mentioned procedures follows.

```
algebra[%nameclash(?C1,?C2)]:-
    if (%nc(?C1,?C2))
    then ( writeln('name clash')@_prolog, true )
    else ( writeln('no name clash')@_prolog, false ).

  %nc(?C1,?C2) :-
      ?C1[?M{?Low:?High}*=>?D],?C2[?M{?Low:?High}*=>?D],!.
  %nc(?C1,?C2) :-
      ?C1[?M(?P){?Low:?High}*=>?D],?C2[?M(?P){?Low:?High}*=>?D],!.

  %nc(?C1,?C2) :- ?C1[?M{?Low:?High}=>?D],?C2[?M{?Low:?High}=>?D],!.
  %nc(?C1,?C2) :- ?C1[?M(?P){?Lo:?Hi}=>?D],?C2[?M(?P){?Lo:?Hi}=>?D],!.
```

If indeed there is a name clash between two **CLASS** instances and a list of specific clashes is required then one can call procedure **%NAMECLASHLIST** in object **ALGEBRA** with the two **CLASS** instances as arguments. The name clashes list, called **?LC1C2SMDMD**, is created in two steps: the first deals with inheritable methods and the second with non-inheritable methods. For inheritable methods a list of pairs, the method name and its return type, is built through a **COLLECTSET** predicate. This list together with the two class instances are passed in to procedure **%MDOVERLAPSTAR** that recursively checks each element in the list for data type signature overlap between the classes. If it is the case the property name is appended. After the inheritable list is checked a similar process is run on non-inheritable properties. Any overlap is appended to the result. The final list is then passed to the main procedure **%NAMECLASHLIST** of object algebra in its third argument.

```
algebra[%nameclashlist(?C1,?C2,?Lc1c2smdmd)]:-
    ?L11=collectbag{?M|?C1[?M*=>?D]},
    ?L12=collectbag{?D|?C1[?M*=>?D]},
    %merge2list(?L11,?L12,[],?Lc1smd),
    %mdoverlapstar(?C1,?C2,?Lc1smd,[],?Lc1c2smd),
    ?L21=collectbag{?M|?C1[?M=>?D]},
    ?L22=collectbag{?D|?C1[?M=>?D]},
    %merge2list(?L21,?L22,[],?Lc1md),
    %mdoverlap(?C1,?C2,?Lc1md,[],?Lc1c2md),
    ?Lc1c2smd[_append(?Lc1c2md)->?Lc1c2smdmd]@_basetype,
    true.
```

```
%mdoverlapstar(?C1,?C2,[?M,?D|?R],?I,?F) :-
  if (?C1[?M{?Low:?High}*=>?D], ?C2[?M{?Low:?High}*=>?D])
  then (%mdoverlapstar(?C1,?C2,?R,[?M,?D|?I],?F))
  else (%mdoverlapstar(?C1,?C2,?R,?I,?F)).
%mdoverlapstar(?_, ?_, [],?I, ?F) :- ?I=?F.


%mdoverlap(?C1,?C2,[?M,?D|?R],?I,?F) :-
  if (?C1[?M{?Low:?High}=>?D], ?C2[?M{?Low:?High}=>?D])
  then (%mdoverlap(?C1,?C2,?R,[?M,?D|?I],?F))
  else (%mdoverlap(?C1,?C2,?R,?I,?F)).
%mdoverlap(?_, ?_, [],?I, ?F) :- ?I=?F.
```

11.2.6 Select

The select by value operator is unary and its output data type signatures are identical to its input. Associated with the select operator's application are a single class instance and a predicate. An instance of the input is part of the query instance, i.e. output, if the predicate evaluation is true for it. It is important to note that the predicate needs to be evaluated on every input argument instance-of and its result is independent of any other instance-of evaluation. There are two issues specific to our data and query model: the first is whether the extent or deep extent of the arguments is required; the second being the demands that this operator is value based and produces a set (i.e. requires duplicate check and elimination).

The following are examples of using the select by value operator, identified by **%SV** in object algebra. The first query sifts for **PERSON** instances whose **FNAME** property is equal to string "**MARY**" (string "**MARY**" is not identifier **MARY**). The first two arguments of the operator list the range and the deep extent requirement. The third argument is the predicate encoded as a functor; the predicate is path expression **FNAME** concatenated to **PERSON** is equal to a string atom whose value is "**MARY**". The result data type signature and the result extent, i.e. single object identified as **MARY**, together with some properties value is given.

```
algebra[%sv(person,"deepextent",eqop(ipe,fname,a(string),"mary"^^_str))].
algquery instance created
signatures created
parsing:
flapply(eqop,ipe,fname,flapply(a,string),_datatype(_string(mary),_string)
)
parsed term (%sbvp_comp_predicate / single_var) :
flapply(eqop,ipe,fname,flapply(a,string),_datatype(_string(mary),_string)
)
data typing:
flapply(eqop,ipe,fname,flapply(a,string),_datatype(_string(mary),_string)
)
data type ok.
filter evaluation
evaluation results (from) to:
[dri,j,joe,jv,mary,michael,nancy,p1,p2,patricia,paul,richard,robert,susan
]
```

```
[mary]
data loid and selection done
data copying done
duplicate list[]
duplicate deletion done
Yes.
?- ?Q:algquery.
?Q = sv(person,"deepextent",_#'18673)
Yes.
?- ?Q:algquery[?M*=>?Dt].
?Q = sv(person,"deepextent",_#'18673)    ?M = fname    ?Dt = string
?Q = sv(person,"deepextent",_#'18673)    ?M = telno    ?Dt = integer
Yes.
?- ?Q:algquery,?I:?Q.
?Q = sv(person,"deepextent",_#'18673)?I =
isv(person,"deepextent",_#'18673,mary)
Yes.
?- ?Q:algquery,?I:?Q[?M->?R].
?Q = sv(…,_#'18673) ?I = isv(…,mary) ?M = fname ?R = "mary"^^_string
?Q = sv(…,_#'18673) ?I = isv(…,mary) ?M = telno ?R = "1234"^^_integer
?Q = sv(…,_#'18673) ?I = isv(…,mary) ?M = telno ?R = "5678"^^_integer
Yes.
```

The following example shows how the select predicate uses logical identifier, i.e. object identifier **MARY**, to filter the **CLASS STUDENT** instance-of collection with. Also note the predicate is unified to a property called **SELECTION** of **ALGQUERY** instance. (Note: all **ISV(…)** identifiers are identical in the following script and therefore the properties appertain to the same object).

```
?- algebra[%sv(student,"deepextent",eqop(loid,mary,loid,self))].
…
Yes.
?- ?_Q:algquery,?I:?_Q,?I[?M->?R].
?I = isv(…)   ?M = enrolon              ?R = bsc_eng
?I = isv(…)   ?M = fname                ?R = "mary"^^_string
?I = isv(…)   ?M = stage                ?R = "g"^^_string
?I = isv(…)   ?M = telno                ?R = "1234"^^_integer
?I = isv(…)   ?M = telno                ?R = "5678"^^_integer
?I = isv(…)   ?M = result("a"^^_string) ?R = idb
?I = isv(…)   ?M = result("b"^^_string) ?R = javaprog
?I = isv(…)   ?M = result("b"^^_string) ?R = pdb
?I = isv(…)   ?M = result("c"^^_string) ?R = dsa
Yes.
?- ?Q:algquery,?Q[?M->?R].
?Q = sv(…)    ?M = selection            ?R = eqop(loid,mary,loid,self)
Yes.
```

The following select by value example shows a predicate over student instances that is a conjunction of conditions: i.e. **STUDENT'S FNAME** is equal to string "**MARY**" and instance identifier is **MARY**.

```
?- algebra[%sv(student,"deepextent",
     and( eqop(ipe,fname,a(string),"mary"^^_string),
          eqop(loid,mary,loid,self) ) ) ].
…
Yes.
?- ?_Q:algquery,?I:?_Q,?I[?M->?R].
?I = isv(…) ?M = enrolon ?R = bsc_eng
?I = isv(…) ?M = fname   ?R = "mary"^^_string
…
Yes.
```

The following is a very common use of select by value in queries and it involves doing a product by value and then a select on their result; specifically it is called an *equi-join*. The first operation is a product between **CLASS** instances **UNIT** and **COURSE**. Both classes have a property called colour, i.e. **CCOLOUR** and **UCOLOUR** respectively, which are assigned strings. The second operation is a select by value and the predicate checks that these two attributes have the same colour value. The example result show that "**OLIVE**" colour is a common value in instances named "**ENGLISH**" and "**FINAL YEAR PROJECT (FYP)**". The third query shows two tuples from the product that satisfies the select predicate.

```
?- algebra[%xv(unit,course,"extent")].
?- algebra[%sv(xv(unit,course,"extent"),"extent",
           eqop(ipe,ccolour,ipe,ucolour))].
…
Yes.
?- ?_Q:algquery,?_Q=sv(?_1,?_2,?_3),
   ?I:?_Q[cname->?Cn,uname->?Un,ucolour->?Uc].
?I=isv(…) ?Cn="computing"^^_s ?Un="dist db"^^_s            ?Uc =
"crimson"^^_s
?I=isv(…) ?Cn="english"^^_s   ?Un="final year project"^^_s ?Uc =
"olive"^^_s
Yes.
```

A join between two ranges that has its predicate not based on equality but any comparison is a called *theta join*.

For the select operator implementation two reified rules are used. These are attached to an object identified with **SVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert a **SV(…)** object is an instance-of **ALGQUERY**. The method takes four arguments: the first one is the collection on which the select predicate is to be executed. The second contains an indicator to state whether a deep extent or an extent is required for the result. The third is unifies the encoded predicate. The fourth has the select by value instance identifier that is uniquely generated for each query instance.

```
svf[algqueryinstance(?C1,?Deep,?Filterlst,?Tag) ->
  ${ (  sv(?C1,?Deep,?Tag):algquery [selection->?Filterlst] :- true  ) }].
```

The second reification concerns the data type signature of the result. Actually the output data type signatures remain intact. The **SVF** object uses the **ALGQUERYDT** property and it takes four parameters: the same set of arguments as **ALGQUERYINSTANCE**. The **ALGQUERYDT** has four instantiations each related to a different properties mode–to cater for inheritable (i.e. ***=>**) and non-inheritable (i.e. **=>**) methods, and arity and non-arity (i.e.

NOT COMPOUND(?M)) methods. Each rule has the same pattern but each fire for a particular data signature type.

```
svf[algquerydt(?C1,?Deep,?Tag) ->
  ${ (  sv(?C1,?Deep,?Tag)[?M{?B:?T}*=>?D] :-
              sv(?C1,?Deep,?Tag):algquery,
              ?C1[?M{?B:?T}*=>?D],
              not compound(?M)@_prolog),
        … ].
```

The object **ALGEBRA** has procedure **%SV** to implement select by value and its invocation requires three variables to be unified. The arguments are the class instance, the *ISA* indicator, and the filtering predicate. This operator has the longest source code listing due to the parser required to read and interpret the predicate.

Procedure **%SV** starts by generating a logical identifier, by invoking **NEWOID{…}** predicate, and unifying it with variable **?TAG**. The two reified rules are invoked, reified, and inserted into the object base – these take care of instance-of assertion (i.e. **ISV(…):SV(…)**) and data type signatures of query instance class. At this point the extent, or deep extent, of the class instance unified to the first argument is created; i.e. **?EXTLST**. The focal part of the implementation is the predicate parser. The procedure that starts the recursive descent parsing is called **%SBVP** and takes six arguments: the first three are identical to the calling procedure **%SV**, the fourth is **?TAG**, the fifth is the extent just generated, and the sixth, **?RESULTLST**, is the list of objects from the extent provided that satisfy the predicate. (The procedure is explained in detail lower down the text). Once parsing and evaluation is the **?RESULTLST** is passed to **%SVDATALOID** to create instance-of assertions and then call procedure **%SVDATA** to move the relative object values from **?C** instance-of objects to **ISV(…)**. Both procedures **%SVDATALOID** and **%SVDATA** follow the same implementation patterns of other operators. The operator ends by checking for value duplicates and purges any present.

```
algebra[%sv(?C1,?Deep,?Filter)] :-
  newoid{?Tag},
  svf[ algqueryinstance(?C1,?Deep,?Filter,?Tag) -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  svf[ algquerydt(?C1,?Deep,?Tag)-> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  if   (?Deep="extent")
  then (?Extlst=
        collectset{?Ileft|?Ileft:?C1,
                          if (?Sc::?C1, ?Ileft:?Sc)
```

```
                              then (false) else (true)})
    else (?Extlst=collectset{?Ileft|?Ileft:?C1}),
    %sbvp(?C1,?Deep,?Filter,?Tag,?Extlst,?Resultlst),
    %svdataloid(?C1,?Deep,?Tag,?Resultlst),
    writeln('data loid and selection done')@_prolog,
    %svdata(?C1,?Deep,?Tag),
    writeln('data copying done')@_prolog,
    algebra[%duplicates(sv(?C1,?Deep,?Tag),?Duplst)],
    write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
    algebra[%delduplicate(?Duplst,sv(?C1,?Deep,?Tag))],
    writeln('duplicate deletion done')@_prolog.
```

The parser and evaluator is the most complex set of procedures in our algebra: it entails, parsing, data type checking, evaluation, and collating results. To leave it as simple as possible the parser and evaluator were designed on a recursive descent parsing mode that unfolds the grammar with successive recursive procedures. There is a procedure for each production rule. Also the evaluation actions are interspersed with and within the production rules. The grammar implemented is given in the following listing (based on basic BNF but has been truncated in depth). The predicate, of which some examples have been given, is a tree structure and Prolog's functors are used. The grammar shows that a predicate is built from a number of conjuncts that use conjunction and disjunction to weave together. Every conjunct, conjunction and disjunction can be negated too. In turn conjuncts are broken down to comparison, *ISA* and instance-of, and "in". The keywords in caps and within inverted commas are functor symbol.

```
search_cond           ::== boolean_term |
                           boolean_term 'OR' search_cond

boolean_term          ::== boolean_factor |
                           boolean_factor 'AND' boolean_term

boolean_factor        ::== 'NOT' boolean_primary |
                           boolean_primary

boolean_primary       ::== predicate |
                           '(' search_cond ')'

predicate             ::== isa_predicate    |
                           exists_predicate |
                           in_predicate     |
                           comp_predicate

exists_predicate      ::== 'EXISTS' '(' simple_subquery ')'

in_predicate          ::== simple_var 'NOT IN' '(' simple_subquery ')' |
                           simple_var 'NOT'    '(' simple_subquery ')' |
                           simple_var 'NOT IN' '(' atom_list ')'       |
                           simple_var 'NOT'    '(' atom_list ')'

atom_list             ::== atom |
                           atom | atom_list

atom                  ::== value   |
                           identifier
```

```
comp_predicate        ::== single_var comparison single_var      |
                           multi_var  comparison multi_var       |
                           single_var comparison simple_subquery |
                           multi_var  comparison simple_subquery

comparison            ::== '=' | '!=' |
                           '>' | '>=' |
                           '<' | '<='

simple_subquery       ::== 'SELECT' select_vars
                           'FROM' expressions

single_var            ::== identifier |
                           path_expression rooted at class |
                           atom

multi_var             ::== path_expression rooted at class  |
                           atom_list
```

The parsing and evaluation routine, **%SBVP**, takes six arguments and all except the last are unified by the calling routine. The evaluation parses the filter and for each of its functors there is a production rule that matches and executes the relation evaluation to filter instances in argument **?EXTLST**; instances that satisfy the predicate are appended to list **?RESULTLST**.

The top most parsing rule is 'search condition', and a procedure named **%SBVP_SEARCH_COND** implements the rule that has two rules: Boolean term or Boolean term in disjunction with search condition (recursive call). At this point the predicate root must have an 'or' functor, i.e. **OR(TERM1,TERM2)** and within its brackets are two terms. Note the calling of **%SBVP_BOOLEAN_TERM** and **%SBVP_SEARCH_COND** with respective terms and the appending of their results. Prior to succeeding with the appended list it is passed through duplicate elimination – the procedure **%LOID_UNIQ** eliminates by logical identifier.

```
%sbvp(?C1,?Deep,?Term,?Tag,?Extlst,?Resultlst) :-
  write('parsing: ')@_prolog,writeln(?Term)@_prolog,
  %sbvp_search_cond(?C1,?Term,?Tag,?Extlst,?Resultlst).

  %sbvp_search_cond(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
    %sbvp_boolean_term(?C1,?Term,?Tag,?Extlst,?Resultlst).
  %sbvp_search_cond(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
    ?Term=or(?Interm1,?Interm2),
    %sbvp_boolean_term(?C1,?Interm1,?Tag,?Extlst,?Resultlst1),
    %sbvp_search_cond (?C1,?Interm2,?Tag,?Extlst,?Resultlst2),
    ?Resultlst1[_append(?Resultlst2)->?New]@_basetype,
    %loid_uniq(?New,[],?Resultlst).
```

The procedure **%SBVP_BOOLEAN_TERM**, on each of its calls a new list of **?EXLIST** is unified, has two rules: Boolean factor or a conjunction of Boolean factors. In the case of a conjunction procedure for Boolean factor and Boolean term are called and the term's functor that identifies it is **AND(…)**. In **%SBVP_BOOLEAN_TERM** that caters for the

AND(TERM1,TERM2) functor the procedures **%SBVP_BOOLEAN_FACTOR** and **%SBVP_BOOLEAN_TERM** (recursive call) are called and their respective results are checked for conjunction by logical identifier. The **%LOID_AND** procedure creates list **?NEW** which is then passed for duplicate elimination by logical identifier.

```
%sbvp_boolean_term(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
  %sbvp_boolean_factor(?C1,?Term,?Tag,?Extlst,?Resultlst).
%sbvp_boolean_term(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
  ?Term=and(?Interm1, ?Interm2),
  %sbvp_boolean_factor(?C1,?Interm1,?Tag,?Extlst,?Resultlst1),
  %sbvp_boolean_term(?C1,?Interm2,?Tag,?Extlst,?Resultlst2),
  %loid_and(?Resultlst1,?Resultlst2,[],?New),
  %loid_uniq(?New,[],?Resultlst).
```

The negation functor is structured as a **NOT(…)** functor. If the predicate, or a part of it, unify with it then **%SBVP_BOOLEAN_PRIMARY** is called and its result is subtracted from **?EXTLST**. The logical identifier based **%LOID_DIFF** procedure takes care of this.

```
%sbvp_boolean_factor(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
  ?Term=not(?Interm),
  %sbvp_boolean_primary(?C1,?Interm,?Tag,?Extlst,?NotResultlst),
  %loid_diff(?Extlst,?NotResultlst,[],?Resultlst).
%sbvp_boolean_factor(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
  %sbvp_boolean_primary(?C1,?Term,?Tag,?Extlst,?Resultlst).
```

A fundamental building block of a selection filter is the comparison operator. It comes into two generic productions: the first is when single value is compared with a single value (subject to comparison data-type conformance); and second is when a value is compared with a set of values (again subject to data-type conformance). The equality operator and the not-equal is applicable to all basic data domains and logical identifiers and in our predicates these are denoted by functors **EQOP(…)** and **NEQOP(…)** respectively. The following example shows a conjunction of two comparison operators by single values. The procedure that takes care of the comparison operators is called **%SBVP_COMP_PREDICATE** and each of its many implementations unifies different version.

```
and(   eqop(ipe,fname,a(string),"mary"^^_string),
       eqop(loid,mary,loid,self) ) )
```

The right rule is chosen according to the first line of the procedure where term, **?TERM**, is unified with, for example, **EPOQ(…)** functor, and if **?COMP** is an acceptable functor, checked with procedure **%SBVP_COMPARISON(…)** then **%SBVP_SINGLE_VAR** is called for the left and right hand side of the comparison. Procedure **%SBVP_SINGLE_VAR** is intended to return the upper cardinality and the data type of the left and right hand side expressions. The

keyword **IPE** in functors indicate a path expression that is built by appending to the **CLASS** instance, **?C1**, the expression in **?ARG2**. For the above first **EQOP** functor example it works as: **STUDENT.FNAME**. The procedure returns upper cardinality '**1**' and string data type for this case. **%SBVP_SINGLE_VAR** handles various type of methods (the usual four types), an 'atom' that is an element of a basic data type, and logical identifier. The next procedure **%SBVP_DD_COMPARISON** evaluates if the comparison operator between the left and right data types is executable. For example while equality and not equal comparison works for all domains and identifiers; on the other hand it does not work for greater than or equal when comparing logical identifiers. The next sequence in the procedure is to check the data typing of the left and right hand side of the comparison and checking that cardinality is singular and data type conformity. If all is well in parsing and data typing the last procedure call, to **%SBVP_COMP_PREDICATE_EVAL_SINGLE,** is made so to evaluate this comparison on every instance. The actual comparison at instance level is done in procedure **%SBVP_COMP_PREDICATE_EVAL_SINGLE_CASE**.

```
  %sbvp_comp_predicate(?C1,?Term,?Tag,?Extlst,?Resultlst) :-
    ?Term=?Comp(?Arg1,?Arg2,?Arg3,?Arg4),
    %sbvp_comparison(?Comp),
    %sbvp_single_var(?C1,?Arg1,?Arg2,?U1,?Dt1),
    %sbvp_single_var(?C1,?Arg3,?Arg4,?U2,?Dt2),
    %sbvp_dd_comparison(?Comp,?Dt1,?CompFlag),
    write('parsed term (%sbvp_comp_predicate / single_var) : ')@_prolog,
    writeln(?Term)@_prolog,
    write('data typing: ')@_prolog,writeln(?Term)@_prolog,
    if   ( ?U1=1,?U2=1,?Dt1=?Dt2,?CompFlag='OK' )
    then ( writeln('data type ok.')@_prolog          )
    else ( writeln('data type error!')@_prolog,
           writeln('- ?U1 ?U2 ?Dt1 ?Dt2 ?CompFlag ')@_prolog,
           writeln(?U1)@_prolog,  writeln(?U2)@_prolog,
           writeln(?Dt1)@_prolog, writeln(?Dt2)@_prolog,
           writeln(?CompFlag)@_prolog,
           false ),
    writeln('filter evaluation')@_prolog,

%sbvp_comp_predicate_eval_single(?C1,?Term,?Tag,?Extlst,[],?Resultlst),
    write('evaluation results (from) to: ')@_prolog,
    writeln(?Extlst)@_prolog,writeln(?Resultlst)@_prolog.

      %sbvp_comparison(eqop) :- true.
      …

      %sbvp_single_var(?C1,?Key,?Arg,?U,?Dt) :-
        not compound(?Arg)@_prolog,
        ?Key=ipe, ?C1[?Arg{?_:?U}*=>?Dt1], dt(?Dt1,?Dt).
      %sbvp_single_var(?C1,?Key,?Arg,?U,?Dt) :-
        compound(?Arg)@_prolog,
        ?Arg=?M(?P),?Key=ipe,?C1[?M(?Pt1){?_:?U}*=>?Dt1],
        dt(?Pt1,?Pt),?P:?Pt,dt(?Dt1,?Dt).
      %sbvp_single_var(?C1,?Key,?I,?U,?Dt)  :-
        ?Key=loid,?I=self,?U=1,?Dt=?C1.

        %sbvp_dd_comparison(eqop, ?_,'OK')    :- true,!.
```

```
        %sbvp_dd_comparison(neqop,?_,'OK')    :- true,!.
        %sbvp_dd_comparison(?Comp,?Dt1,?Flag) :-
           if   (?Comp=gtop;?Comp=gteop;?Comp=ltop;?Comp=lteop)
           then ( if   ( ?Dt1=_integer;?Dt1=_decimal;?Dt1=_string )
                  then ( ?Flag='OK',true   )
                  else ( ?Flag='ERR',false )  )
           else ( ?Flag='ERR',false,!).

           %sbvp_comp_predicate_eval_single
               (?C1,?Term,?Tag,[?I|?Extlst],?SResultlst,?FResultlst) :- …
.

           %sbvp_comp_predicate_eval_single_case
             (?C1,eqop,?Arg1,?Arg2,?Arg3,?Arg4,?I)  :-
                 %sbvp_cpesc_v(?Arg1,?Arg2,?I,?Left),
                 %sbvp_cpesc_v(?Arg3,?Arg4,?I,?Right),
                 if   ( ?Left=?Right  )
                 then ( true          )
                 else ( false         ).
```

The above example deals with simple comparison; but the parser deals with parsing and evaluation of predicates that contain singleton to set comparison too. For example, implementation of universal and existential quantification over set value properties follows:

```
// reification rules for equality, and greater and less than comparisons

equery [test -> ${(totest(?Head,?Value):-?Head = ?Value)}].
gtquery[test -> ${(totest(?Head,?Value):-?Head @>= ?Value)}].
ltquery[test -> ${(totest(?Head,?Value):-?Head @=< ?Value)}].
…

//  universal quantification and existential quantification routines

forall([],            ?_Query,?_Value  )  :- true.
forall([?Head|?List], ?Query, ?Value   )  :-
    if ( ?Query[test -> ?Rules], insertrule{?Rules}, totest(?Head,?Value)
)
    then ( forall(?List,?Query,?Value) )
    else ( false ).

forsome([],            ?_Query,?_Value  ) :- false.
forsome([?Head|?List], ?Query, ?Value   ) :-
    if   ( ?Query[test -> ?Rule],insertrule{?Rule},totest(?Head,?Value) )
    then ( true )
    else ( forsome(?List,?Query,?Value) ).
```

The first sample of reification above, attached to instances **EQUERY** for example, deals with simple comparison based on identifier equality or basic domain comparison. Universal and existential quantification are catered for by rules whose heads are denoted by **FORALL** and **FORSOME** respectively. The following script shows a high-level use of these generalised quantification operations. The first query's purpose is to display the sample data for each **STUDENT** instance and their **GRADES** – note that the conversion of a set into a list does not change the generality of the solution. The second query fires for **STUDENT** instances whose **GRADES** collated have a universal quantification that requires all of the **GRADES** are string **'C'**. The third query, similar to the second, asks for **STUDENT** instances who's **GRADES** are

all string greater than or equal string **'C'**. (Note: string **'D'** is greater than string **'C'**).

The forth query sifts for **STUDENT** instances that have at least one **GRADE 'C'**.

```
?- ?S:student, ?Sglst=collectset{?W|?S[result(?W)->?G]}.
?S = mary      ?Sglst = ["a"^^_string, "b"^^_string, "c"^^_string]
?S = michael   ?Sglst = ["c"^^_string]
?S = nancy     ?Sglst = ["b"^^_string, "d"^^_string]
?S = patricia  ?Sglst = ["a"^^_string, "c"^^_string, "f"^^_string]
?S = paul      ?Sglst = ["b"^^_string
?S = richard   ?Sglst = ["d"^^_string, "f"^^_string]
?S = robert    ?Sglst = ["c"^^_string]
?S = susan     ?Sglst = ["a"^^_string]
Yes.
?- ?S:student, ?Sglst=collectset{?W|?S[result(?W)->?G]},
   forall(?Sglst,equery,"c"^^_string).
?S = michael    ?Sglst = ["c"^^_string]

?S = robert    ?Sglst = ["c"^^_string]
Yes.
?- ?S:student, ?Sglst=collectset{?W|?S[result(?W)->?G]},
   forall(?Sglst,gtquery,"c"^^_string).
?S = michael   ?Sglst = ["c"^^_string]
?S = richard   ?Sglst = ["d"^^_string, "f"^^_string]
?S = robert    ?Sglst = ["c"^^_string]
Yes.
?- ?S:student, ?Sglst=collectset{?W|?S[result(?W)->?G]},
   forsome(?Sglst,equery,"c"^^_string).
?S = mary      ?Sglst = ["a"^^_string, "b"^^_string, "c"^^_string]
?S = michael   ?Sglst = ["c"^^_string]
?S = patricia  ?Sglst = ["a"^^_string, "c"^^_string, "f"^^_string]
?S = robert    ?Sglst = ["c"^^_string]
Yes.
```

The parsing and evaluation coding of predicates in a select-by-value statement is the longest and most demanding; demanding in terms of coding innovation (albeit this is a qualitative measure). Nonetheless, even if kept somehow limited in range when compared to ODMG OQL predicates, the coding and implementation through Flora-2 does not indicate it cannot suffice.

Properties of the Select operator

As in most of the operators, select by value is lossy. Every relational operator has a number of algebraic properties: for example select by value is commutative, and idempotent.

$$\sigma_{\theta 1}\big(\sigma_{\theta 2}(E)\big) \Leftrightarrow \sigma_{\theta 2}\big(\sigma_{\theta 1}(E)\big)$$

Also a sequence of select predicates is equal to their conjunction.

$$\sigma_{\theta 1 \wedge \theta 2}(E) \Leftrightarrow \sigma_{\theta 1}\big(\sigma_{\theta 2}(E)\big)$$

We can easily follow the pattern used earlier for commutativity by introducing appropriate reified rules. But at this point it is best to see the differences here. Firstly, the commutativity previously encountered was between binary operators; in this case it is commutativity in the predicate conjunctive parts. Secondly, it is a tenable assumption that the string of conjuncts in a predicate is more than a few (i.e. the more numerous are the number of properties in the query extent the more conjuncts to expect). If, for the sake of simplicity, a selection predicate has only predicates in a conjunct combination then the selection predicate is much like a binary tree with **`AND()`** in root and inner nodes of the tree. The number of permutations in a binary tree through conjunction commutativity and cascading of select operations grows most aggressively with the number of predicates. These numbers are within the genera of numbers called Catalan Numbers and, for example, seven predicates have 429 permutations and thirteen predicates have 742900 permutations. An *associahedron* for four predicates is given here in figure 12.1 below. In view of this it is best to keep re-ordering of predicates as a later part of query optimisation rather than at the initial structural equivalence part. Re-ordering in later parts of query optimisations is common in DBMSs with algebraic like access plans:

Oracle { WWW.ORACLE.COM/US/PRODUCTS/DATABASE/OVERVIEW/INDEX.HTML };

DB2 { WWW-01.IBM.COM/ SOFTWARE/ DATA/DB2 };

MS SQL Server { WWW.MICROSOFT.COM/EN-US/SQLSERVER/PRODUCT-INFO.ASPX }; and

PostgreSQL { WWW.POSTGRESQL.ORG }.

Select by value has a number of equivalence relationships with other operators too.

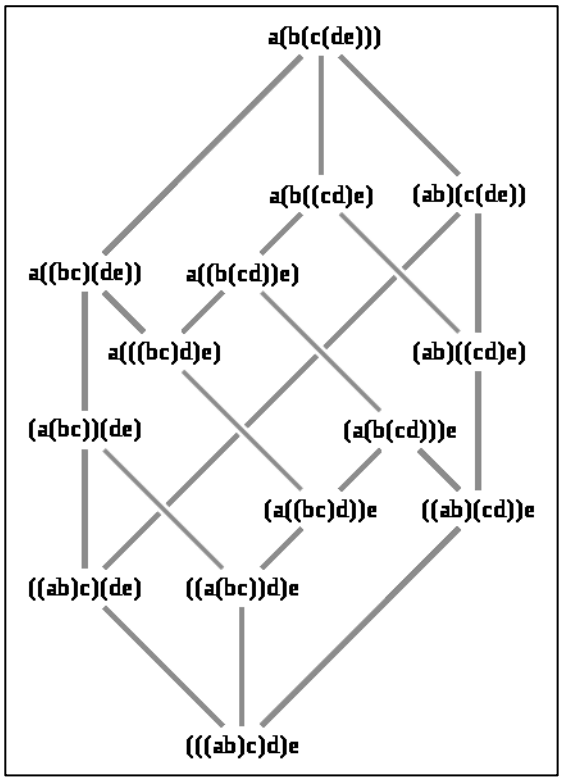


Figure 12.1: The different permutations of 4 predicates – diagram is called an *associahedron* and is credited to David Eppstein { WWW.ICS.UCI.EDU/~EPPSTEIN }.

11.3 Summary

In this chapter we provided details of our query model, some algebraic operators, and how these are integrated and aided by our framework. In the next chapter we continue with the other five operators. Consequently we leave a summary of the operators here for the next chapter (section 12.3).

Chapter 12

Object-Oriented Query Model

(II)

12 Object-Oriented Query Model (II)

This chapter continues from the previous chapter in describing algebraic operators that form part of our query model. In this chapter the operators' origin is not really relational, although such operators have been advocated for the relational model. In fact their origin is more from the functional and nested model.

12.1 Object Algebra and an Object Framework (II)

The five operators described here are represented and have been implemented in a similar method as was used in the previous chapter. Also the interaction with the underlying framework is similar.

12.1.1 Map and Aggregate

The operators introduced so far do not offer any possibility to compute functions on objects in the collections. Neither is it possible to restructure the query instance data-type signature outside the input ranges. The unary operators map and aggregate address this deficiency. The map operator applies a function to each object's property value and returns another; for example a map function on **CLASS STUDENT** is converting each **FNAME** property value into its upper case. On the other hand the aggregate operator collects object instances that share a common value for a property and applies a function collectively on this partition; for example an aggregate function on **CLASS STUDENT** for each distinct **STAGE** property value is counting the instances in each **STAGE** partition. In our implementation projection is included in the operator's implementation.

12.1.1.1 Map

The unary map algebraic operator applies a function, passed in as an argument, to an indicated object property of every instance-of a class. The data type of the function's domain must be type compatible with the data type of the property it is being applied to. Path expression traversal is considered as a function. There are two issues specific to our data and query model: the first is whether the extent or deep extent of the arguments is required; the second being the demands that this operator is value based and produces a set (i.e. requires duplicate check and elimination).

In the first two queries of the following script we first map a string into its reverse and in the second we leave the output the same as the input: the mapping functions are called **REV** and

IDENTITY respectively and defined through rules in the framework. The third query shows the data-type signature of properties **FNAME** and **TELNO** in **CLASS STUDENT**. The forth query shows the application of the algebraic map operator on **STUDENT'S** deep extent two properties, **FNAME** and **TELNO**, with the **REV** and **IDENTITY** mappings. The data type signature of the query instance show two attributes named **IDENTITYTELNO** and **REVFNAME**; clearly the respective mapping functions are not in data-type error. The last query shows the data output of the query instance.

```
?- ?Msg="Hello world"^^_string, rev(?Msg,?Rmsg).
?Msg = "Hello world"^^_string   ?Rmsg = "dlrow olleH"^^_string
Yes.
?- ?Msg="Hello world"^^_string, identity(?Msg,?Imsg).
?Msg = "Hello world"^^_string   ?Imsg = "Hello world"^^_string
Yes.
?- student[fname*=>?Dt1;telno*=>?Dt1].
?Dt1 = integer
?Dt1 = string
Yes.
?- algebra[%wv(student,"deepextent",[fname,telno],[rev,identity])].
method not found list: []
Methods exist: ok
map not found list: []
Maps exist: ok
Method dtype & Maps dtype match: OK
algquery instance created
signatures created
data loid and move done
duplicate deletion done
Yes.
?- ?Q:algquery,?Q[?M*=>?Dt].
?Q = wv(student,"deepextent",_#'19684)  ?M = identitytelno   ?Dt = integer
?Q = wv(student,"deepextent",_#'19684)  ?M = revfname        ?Dt = string
Yes.
?- ?_Q:algquery,?I:?_Q[?M->?V].
?I = iwv(…,mary)     ?M = identitytelno  ?V = "1234"^^_integer
?I = iwv(…,mary)     ?M = identitytelno  ?V = "5678"^^_integer
?I = iwv(…,mary)     ?M = revfname       ?V = "yram"^^_string
?I = iwv(…,michael)  ?M = identitytelno  ?V = "3456"^^_integer
…
Yes.
```

In a later part of this section another example of using the map operand to traverse path expressions is given.

For the map operator's implementation two reified rules are used. These are attached to an object identified with **WVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert a **WV(…)** object is an instance-of **ALGQUERY**. The method takes five arguments: the first two are the collections on which the mapping is to be executed and an indicator to state whether a deep extent or an extent is required for the result. The third and fourth contains two equi-numerous lists – one with property names to map and the other functions to apply. The fifth argument is a

unique identifier. Note that object **WV(…)** instantiated has two properties that unify with the list of properties and the functions applied; called **PROPERTY** and **MAP**.

```
wvf  [algqueryinstance(?C1,?Deep,?Usmthlst,?Usmaplst,?Tag) ->
  ${ (  wv(?C1,?Deep,?Tag):algquery
        [property->?Usmthlst, map->?Usmaplst] :- true  ) }].
```

Before we describe the second reification definition it is important to describe necessary details appertaining to mapping functions. We need to encode their name, instance-of a **MAP**, data type signature, and the implementation of the mapping function. An object **MAP** is defined in the framework and has three data type signatures: the first to record the mapping's **SOURCE**, the second to record the mapping's **TARGET** data type, and the third to record the mapping's **FUNCTION** name. A mapping function is defined as an instance of object **MAP**. The mapping functions used previously are in fact an instance-of **MAP**, and for example object **REV** has the same function name. The implementation of **REV** is found in predicate **MAPF** with **REV** as its first argument.

```
map[ source*=>object, target*=>object, function*=>string ].

rev:map     [ source->string,   target->string,  function->rev      ].
identity:map[ source->_object,  target->_object, function->identity ].
incone:map  [ source->integer,  target->integer, function->incone   ].

mapf(rev,      ?S, ?R) :- ?S[_reverse->?R]@_basetype.
mapf(identity, ?V, ?R) :- ?V=?R.
mapf(incone,   ?V, ?R) :- ?R is ?V+1.
```

The second reification concerns the building of rules for creating data type signature of the result. The data type signature of the query result is mainly determined by the target data type of the involved mapping functions. Also required is the creation of a name for each new property: in this case it was elected that the **MAP** instance name is made a prefix of the original property name. So for example applying **REV** to **FNAME** produces the name **REVFNAME**. The data type reification starts by sifting for the relative **ALGQUERY** instance of, i.,e. **WV(…)**. For each property in the method list bound to **?USMTHLST** we search for all non-compound inheritable methods and bind it to **?M**. For this method we find its corresponding mapping function, i.e. **?MAPM**, in the list bound to variable **?USMAPLST**. Details of the mapping are found in the relative instance-of **MAP**; i.e. the **TARGET** property holds the data type of the mapping result. The last action is the creation of a new name for the mapped property. At this point the data-type signature derived from **?M**

can be asserted. There are, as usual, four rules based on the property's mode (e.g. inheritable or not) and arity.

```
wvf  [algquerydt(?C1,?Deep,?Usmthlst,?Usmaplst,?Tag) ->
  ${ (  wv(?C1,?Deep,?Tag)[?Nm{?B:?T}*=>?Md]  :-
          wv(?C1,?Deep,?Tag):algquery,
          %lmember(?M,?Usmthlst),
          ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog,
          %lookup121(?Usmthlst,?Usmaplst,?M,?Mapm),
          ?Mapm:map[target->?Md2],
          if   (?Md2=_object)
          then (?Md=?D)
          else (?Md=?Md2),
          addprefix(?Mapm,?M,?Nm)  ), … ].
```

The actual evaluation of mapping by value is done through procedure **%UV** in the object **ALGEBRA**. The procedure **%UV** takes four arguments: the first two are the classes' instances and an indicator on the extent required (i.e. unified to variable **?DEEP**). The last two arguments are equi-numerous lists of method names and mapping functions. The first part of the implementation deals with pre-checks: for example both methods and mapping list have their content correct – i.e. properties and map instances exist. A significant check, in procedure **%WVDATACHECKMAPLST**, needs to ensure that the data type of a method is compatible with the source data type of mapping function being applied over it. The procedure is recursive and uses data type signatures from the **CLASS** extent of the query and the relative **MAP** instance-of. The actual processing starts by creating a unique identifier, unified with **?TAG**, and the reifications of the query instance and the query's data type signature. The next procedure call, to **%WVDATALOID**, is very similar to previous operators' implementation. On the other hand the implementation of **%WVDATA** is more involved but follows the sequence of logical steps to create the map-query results data-type signature. Once the instances are created than the query instance collection is checked for duplicates and if need be duplicate copies are purged.

```
algebra[%wv(?C1,?Deep,?Usmthlst,?Usmaplst)] :-
  algebra[%methodsexist (?C1,?Usmthlst,?Nomthlst)],
  if ( ?Nomthlst = [] ) …
  algebra[%mapsexist(?Usmaplst,?Nomaplst)],
  if ( ?Nomaplst = [] ) …
  if ( not %wvdatacheckmaplst(?C1,?Usmthlst,?Usmaplst,?_Comment) )
  then (writeln('Method dtype & Maps dtype match: Error!')@_prolog, fail)
  else (writeln('Method dtype & Maps dtype match: OK')@_prolog),
  newoid{?Tag},
  wvf[ algqueryinstance(?C1,?Deep,?Usmthlst,?Usmaplst,?Tag) -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  wvf[ algquerydt(?C1,?Deep,?Usmthlst,?Usmaplst,?Tag)-> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  %wvdataloid(?C1,?Deep,?Tag),
  %wvdata(?C1,?Deep,?Usmthlst,?Usmaplst,?Tag),
  writeln('data loid and move done')@_prolog,
```

```
algebra[%duplicates(wv(?C1,?Deep,?Tag),?Duplst)],
write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
algebra[%delduplicate(?Duplst,wv(?C1,?Deep,?Tag))],
writeln('duplicate deletion done')@_prolog.
```

The procedure **%WVDATA**, called from the main procedure **%WV**, moves data from the query instance source, i.e. bound to **?C1**, to instances-of **WV(…)**. For each property in the method list bound to **?USMTHLST** we search for all non-compound inheritable methods and bind them to **?M**. For this method we find its corresponding mapping function, i.e. **?MAPM**, in the list bound to variable **?USMAPLST**, through procedure call to **%LOOKUP121**. The next action is the creation of a new name for the mapped property – i.e. bound to **?NM**. The final part is the application of the mapping function to the original values; where the original identifier, bound to **?I**, is part of the query instance-of identifier. The actual mapping is done through procedure **MAPF** that given the mapping function and original value return the mapping – i.e. bound to **?V**. This data is passed to usual procedure to insert the assertion. This procedure has to be repeated for the other three types of methods, e.g. inheritable or non-inheritable.

```
%wvdata(?C1,?Deep,?Usmthlst,?Usmaplst,?Tag):-
  ?LeftIsmlst=
  collectset{ ?Wl | wv(?C1,?Deep,?Tag):algquery,
                    %lmember(?M,?Usmthlst),
                    ?C1[?M{?B:?T}*=>?_D],
                    not compound(?M)@_prolog,
                    %lookup121(?Usmthlst, ?Usmaplst, ?M, ?Mapm),
                    ?Mapm:map,
                    addprefix(?Mapm,?M,?Nm),
                    iwv(?C1,?Deep,?Tag,?I)[],
                    ?I:?C1[?M->?R],
                    mapf(?Mapm,?R,?V),
                    ?Wl=f(?C1,?Deep,?Tag,?I,?Nm,?V) },
  %wvinsertsmethod(?LeftIsmlst), … .

    %wvinsertsmethod([?H|?Rlst]):-
      ?H=f(?C1,?Deep,?Tag,?I,?M,?R),
      insert{ iwv(?C1,?Deep,?Tag,?I)[?M->?R] },
      %wvinsertsmethod(?Rlst).
    %wvinsertsmethod([]).
```

Map and path expressions

The map operator is useful to implement expressions involving path expression too. For example if a range contains a property whose value is a logical identifier it is advantageous to be able traverse to it rather than join the two ranges. In the following example we are interested to retrieve a student's course name (i.e. **CNAME**), through its **ENROLON** property. The first query works out the data type signature of the path expression (i.e. **STUDENT.ENROLON.CNAME**). The second query shows the results of applying the same path expression on **STUDENT'S** instances.

The third query shows how the same result, as the second query, can be achieved through a user-defined map function; e.g. called **STUDCNAME**.

```
?- student[enrolon *=> ?Dt1[cname *=> ?Fdt]].
?Dt1 = course      ?Fdt = string
Yes.
?- ?I:student, ?I.enrolon.cname=?R.
?I = mary          ?R = "engineering"^^_string
…
Yes.
?- ?I:student,mapf(studcname,?I.enrolon,?R).
?I = mary          ?R = "engineering"^^_string
…
Yes.
```

Consequently if we want to output the student's name and the name of the course she is following then we map **FNAME** and **ENROLON** with **IDENTITY** and **STUDCNAME** for **STUDENT'S** instances of interest.

```
?- algebra[%wv(student,"deepextent",[fname,enrolon],[identity,studcname])].
```

The map function **STUDCNAME** definition follows:

```
studcname:map[ source->integer, target-> string, function->studcname].

mapf(studcname, ?V, ?R) :- ?V:course, ?V[cname->?R].
```

In our framework there are data-type routines to verify a path expression existence in our schema and work out its full data type characteristics. These should be used in the definition of any mapping function over and above the data-type checks inbuilt in the data type construction of the mapping operand. Also it is expected that when asserting an instance-of map its **SOURCE** and **TARGET** properties are worked out.

Properties of Map operator

The main property of mapping is its functional composition; that if g and f are mappings then their composition is equal to their sequential application (class denotes ranges – e.g. **CLASS** instance).

$$(g\ o\ f)(class) = g\left(f(class)\right)$$

In general composition is associative but not commutative.

12.1.1.2 Aggregate

The unary aggregate operator first partitions the extent on an indicated property's distinct values and then applies an aggregate function to a specified property for every partition. The aggregate functions supported are sum, minimum, maximum, and count; also all except count expect the function to be applied to a numeric property. The output data type of all aggregate functions is

numeric. There is an issue specific to our data and query model: i.e. whether the extent or deep extent of the arguments is required.

The following examples are based on a class called **TRAN** (for transaction) whose property's data type signatures are listed in the first example query. Properties **QTY**, **SALE** and **COST** are for used as input to aggregate functions, and properties **MONTH**, **YEAR** and **UNITCHARGE** are used to partition the extent. (Note this is an arbitrary selection for example queries).

The second example query below shows a simple aggregate operation over the deep extent of class **TRAN**. The partitioning is being requested on a single property, i.e. **YEAR**. That is the query deep extent is 'converted' into partitions based on unique values of **YEAR** property values. For example if **TRAN'S YEAR** property projection yields three distinct year values then a partition is built for each of these. On each of these partitions the operator applies the aggregate function maximum, i.e. **MAX**, on all of the partition's **SALE** property values. The interesting bit is the data-type signature of the query instance: the third query shows that it has two data-type signatures. There is a data-type signature for the partitioning attribute, in this case **YEAR**, and another that holds the result of applying the **MAX** aggregate function to property **SALE**, it is called **MAXSALE**. The last query shows the query instance extent on test data.

```
?- tran[?M*=>?Dt].
?M = cost      ?Dt = integer
?M = month     ?Dt = integer
?M = qty       ?Dt = integer
?M = sale      ?Dt = integer
?M = unitchg   ?Dt = unit
?M = year      ?Dt = integer
Yes.
?- algebra[%gv(tran,"deepextent",[year],[sale],[max])].
Aggregate arguments match: ok
method not found list: []
Methods exist: ok
method not found list: []
Agg. Methods exist: ok
aggregate not found list: []
Aggregates exist: ok
algquery instance created
signatures created
aggregate loid and data ok.
Yes.
?- ?Q:algquery[?M*=>?Dt].
?Q = gv(tran,"deepextent",…)    ?M = maxsale   ?Dt = double
?Q = gv(tran,"deepextent",…)    ?M = year      ?Dt = integer
?- ?_Q:algquery,?I:?_Q[year->?Y,maxsale->?M].
?I=igv(…)   ?Y = 2010  ?M = 400
?I=igv(…)   ?Y = 2011  ?M = 550
?I=igv(…)   ?Y = 2012  ?M = 550
Yes.
```

The following example is a more ambitious use of the aggregate operator. For example the partitioning request is now a combination of three properties, i.e. for the projection of **YEAR**, **MONTH**, and **UNITCHG**, and two aggregates are requested on each partition namely **SUM** on each property cost and sale. The query for the data type signature of the query instance show that it has five signatures: three for the partitioning properties and one each for the aggregate function. The last query shows the values assigned to property **SUMCOST**; e.g. partition for **2010**, **JANUARY** (i.e. **1**), and **IDB** has a summed cost of 10.

```
?- algebra[%gv(tran,"deepextent",[year,month,unitchg],[cost,sale],[sum,sum])].
Aggregate arguments match: ok
method not found list: []
Methods exist: ok
method not found list: []
Agg. Methods exist: ok
aggregate not found list: []
Aggregates exist: ok
algquery instance created
signatures created
signatures created
aggregate loid and data ok.
?- ?Q:algquery,?I:?Q.
?Q = gv(tran,"deepextent",…)    ?I = igv(tran,"deepextent",…,2010,1,idb)
?Q = gv(tran,"deepextent",…)    ?I = igv(tran,"deepextent",…,2010,2,pdb)
…
?- ?Q:algquery,?Q[?M*=>?Dt].
?Q = gv(tran,"deepextent",…)  ?M = month    ?Dt = integer
?Q = gv(tran,"deepextent",…)  ?M = sumcost  ?Dt = double
?Q = gv(tran,"deepextent",…)  ?M = sumsale  ?Dt = double
?Q = gv(tran,"deepextent",…)  ?M = unitchg  ?Dt = unit
?Q = gv(tran,"deepextent",…)  ?M = year     ?Dt = integer
Yes.
?- ?Q:algquery,?I:?Q[sumcost->?A].
?Q = gv(tran,"deepextent",…)   ?I = igv(…,2010,1,idb)       ?A = 10
?Q = gv(tran,"deepextent",…)   ?I = igv(…,2010,2,pdb)       ?A = 10
?Q = gv(tran,"deepextent",…)   ?I = igv(…,2010,3,ddb)       ?A = 20
?Q = gv(tran,"deepextent",…)   ?I = igv(…,2010,4,dsa)       ?A = 20
?Q = gv(tran,"deepextent",…)   ?I = igv(…,2010,5,javaprog)  ?A = 40
?Q = gv(tran,"deepextent",…)   ?I = igv(…,2010,6,softeng)   ?A = 40
…
Yes.
```

For the map operator implementation two reified rules are used. These are attached to an object identified with **GVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert that **GV(…)** object is an instance-of **ALGQUERY**. The method takes six arguments: the first two are the collections on which the mapping is to be executed and an indicator to state whether a deep extent or an extent is required for the input. The third parameter is a list of property names, of **CLASS** instance unified in the first argument, on which partitioning is to be done. The fourth and fifth contains two equi-numerous lists – one with property names on which an aggregate is to be executed, and the other is the aggregate functions to apply. The sixth argument is a unique

identifier. Note that object **GV(…)** instantiated has three properties that unify with the list of properties to partition with, the list of properties to apply the aggregates to, and the aggregate functions applied.

```
gvf  [algqueryinstance(?C1,?Deep,?Usmthlst,?Aggmthlst,?Aggfunclst,?Tag) ->
  ${ (  gv(?C1,?Deep,?Tag):algquery
        [groupby  -> ?Usmthlst,
         aggoutput-> ?Aggmthlst,
         funcaggout->?Aggfunclst] :- true  ) }].
```

The second reification concerns the building of rules for creating data type signature of the result. The data type signature of the query result is mainly determined by the partitioning properties and the target data type of the involved aggregate functions. Also required is the creation of a name for each new property: in this case it was elected that the aggregate function name is made a prefix of the original property name. The data type reification starts by sifting for the relative **ALGQUERY** instance of, i.,e. **GV(…)**. For each partitioning property in the method list bound to **?USMTHLST** we search for all non-compound inheritable methods and bind it to **?M**. The rule simply copies their data-type signatures. In the case of the aggregate properties we start as for partitioning properties but then investigate the aggregating function to work out its data type and a new property name. For this method we find its corresponding aggregate function, i.e. **?MAAGF**, in the list bound to variable **?AGGFUNCLST**. The last action is the creation of a new name for the mapped property. At this point the data-type signature that is derived from **?M** can be asserted; note that its data type is **DOUBLE** as all aggregate functions allowed return a **DOUBLE**. There are, as usual, four rules based on the property's mode (e.g. inheritable or not) in two pairs; i.e. the aggregating and functional methods.

```
gvf[  algquerydt(?C1,?Deep,?Usmthlst,?Aggmthlst,?Aggfunclst,?Tag)  ->
   ${ // group by attributes
      (  gv(?C1,?Deep,?Tag)[?M{?B:?T}*=>?D]     :-
            gv(?C1,?Deep,?Tag):algquery,
            %lmember(?M,?Usmthlst),
            ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog ),
           …,
     // aggregate functions attributes (all numeric ie double)
     (  gv(?C1,?Deep,?Tag)[ ?Maggr{1:1}*=>double]     :-
              gv(?C1,?Deep,?Tag):algquery,
             %lmember(?M,?Aggmthlst),
             ?C1[?M*=>?_D], not compound(?M)@_prolog,
             %lookup121(?Aggmthlst,?Aggfunclst,?M,?Maggf),
             addprefix(?Maggf,?M,?Maggr)
     ), …
     } ].
```

The actual evaluation of aggregation by value is done through procedure **%GV** in object **ALGEBRA**. The procedure **%GV** takes five arguments: the first two are the classes' instances and an indicator

on the extent required (i.e. unified to variable **?DEEP**). The third is a list of properties, from the class instance bound in the first argument, on which partitioning is to be done. The last two arguments are equi-numerous lists of method names and aggregating functions. The first part of the implementation deals with pre-checks: a) there needs to be at least one partitioning property; b) for every property to aggregate there needs to be one, and only one, aggregating function; c) the list of aggregating properties is made up of methods present in the class being aggregated and they are single valued – **ALGEBRA** object has procedure **%SINGLEMETHODSEXIST** to check this; d) the properties being aggregated must be present and singular (not sets); e) the functions listed in the aggregate function list are instances of the ones allowed. After the checks succeed a unique identifier is generated and the two reification rules are instantiated and inserted into the object base. These rules assert a query instance (i.e. **GV(…):ALGQUERY[…]**) and the data type signature of the query instance. Since the output of the aggregate operator is not 1-1 to the extent instances then the instance–of and actual copying is done through a single procedure called **%GVAGGLOIDDATA**. The procedure takes the same arguments as the operator plus the unique identifier generated (i.e. bound to **?TAG**).

```
algebra[%gv(?C1,?Deep,?Usmthlst,?Aggmthlst,?Aggfunclst)] :-
  ?Nigb =count{?Igb | %lmember(?Igb, ?Usmthlst)   },
  ?Niae =count{?Iae | %lmember(?Iae, ?Aggmthlst)  },
  ?Niaf =count{?Iaf | %lmember(?Iaf, ?Aggfunclst) },
  if   ( ( ?Nigb > 0 ), ( ?Niae=?Niaf )  )  … ,
  algebra[%singlemethodsexist (?C1,?Usmthlst,?Nousmthlst)],
  if ( ?Nousmthlst = [] ) … ,
  algebra[%singlemethodsexist(?C1,?Aggmthlst,?Noaggmthlst)],
  if ( ?Noaggmthlst = [] ) … ,
  algebra[%aggsexist(?Aggfunclst,?Noaggfunclst)],
  if ( ?Noaggfunclst = [] ) … ,
  newoid{?Tag},
  gvf[ algqueryinstance(?C1,?Deep,?Usmthlst,?Aggmthlst,?Aggfunclst,?Tag)
       -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  gvf[ algquerydt(?C1,?Deep,?Usmthlst,?Aggmthlst,?Aggfunclst,?Tag)
       -> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  //generate aggregate loid and data from partitioning
  if ?Deep="deepextent" then (?Deepc='deepextent') else (?Deepc='extent'),
  %gvaggloiddata(?C1,?Deepc,?Usmthlst,?Aggmthlst,?Aggfunclst,?Tag),
  writeln('data loid and aggregate done')@_prolog.
```

Procedure **%GVAGGLOIDDATA** is tedious if not straightforward. The procedure needs to build a rule that implements the aggregation function on the partitioning properties provided for every instance of the aggregating function list. The procedure needs to build the syntax of the rule on the lines shown in the following template.

```
?Agg = aggfunction { property_to_agg [ partitioning_methods+ ]|
                  ?I:class_extent,
                  ?I[property_to_agg,
                     partitioning_methods+]
                 }
```

To help out with the string construction the procedure uses three other procedures to help with concatenation (e.g. **SC(..)** with various arities, **ML2SC**, and **MV12SC**). Rather than using reification to instantiate and update the object base one has to pass to the underlying engine the rule, in string format, to parse and then evaluate. The reason is that coding the various combinations of inputs in lists with reification is not possible. Flora-2 has a special predicate in module parse to read a string and after parsing it evaluates it in the object base, i.e. **%READALL(…)@_PARSE**. In procedure **INSERTRULEBYPARSEANDEVAL(…)** this **%READALL(…)** predicate is called.

```
%gvaggloiddata(?C1,?Deep,?Usmthlst,[?Hmth|?Aggmthlst],[?Hf|?Aggfunclst],?Tag)
  :-  sc('"',?Deep,'"',?Deepe),
      ?T0='igv(', ?L1=[?C1,?Deepe,?Tag],l2sc(?L1,?T0,?T1),
      sc(?T1,',',?T2),
      ml2sc(?Usmthlst,?T2,?T3),
      ?T4='):gv(', sc(?T3,?T4,?T5),
      ?L2=[?C1,?Deepe,?Tag],l2sc(?L2,?T5,?T6),
      ?T7=')[',sc(?T6,?T7,?T8),
      sc(?T8,?Hf,?Hmth,?T9),
      sc(?T9,'->?Agg] :- ?Agg = ',?Hf,?T10),
      sc(?T10,' {?',?Hmth,?T11),
      sc(?T11,'[',?T12),
      ml2sc(?Usmthlst,?T12,?T13),sc(?T13,']|?I:',?C1,',?I[',?T14),
      mvl2sc(?Usmthlst,?T14,?T15),
      sc(?T15,',',?Hmth,'->?',?Hmth,?T16),
      sc(?T16,']}.',?T17),
      writeln('')@_prolog,
      writeln(?T17)@_prolog,
      insertrulebyparseandeval( ?T17 ),
      %gvaggloiddata(?C1,?Deep,?Usmthlst,?Aggmthlst,?Aggfunclst,?Tag).
  %gvaggloiddata(?_C1,?_Deep,?_Usmthlst,[],[],?_Tag).
```

The following listing shows a rule constructed for one case of aggregating function.

```
Remark    Example query
Remark    algebra[%gv(tran,"deepextent",
Remark            [year,month,unitchg],[cost,sale],[sum,sum])].

Remark    Rule generated for cost by year, month & unitchg and sum aggregate

igv(tran,"deepextent", …,?year,?month,?unitchg):
 gv(tran,"deepextent", … )[sumcost->?Agg]
  :- ?Agg = sum {?cost[?year,?month,?uitchg]|
                 ?I:tran,
                 ?I[year->?year,
                    month->?month,
                    unitchg->?unitchg,
                    cost->?cost]
                },
```

Properties of Map and Aggregate operators

An interesting property of the sequence application of the same aggregate operator, given that the partitioning property lists are in a sequential subset relationship too, is that the sequence is equal to the last application (and *vice versa*).

$$\sum_{A_n}\left(\sum_{A_{n-1}}\cdots\left(\sum_{A_1}(class)\right)\cdots\right)=\sum_{A_n}(class)$$

$$where\ A_n \subseteq A_{n-1} \subseteq \cdots \subseteq A_1,$$

$$and\ \sum\ is\ the\ same\ aggregate\ function.$$

Another useful observation is some aggregate functions are derivable from other aggregates. For example the aggregate average is equal to aggregate sum divide by aggregate count.

12.1.2 Nest and Unnest

The nest and unnest by value operands are capable of building composite objects and flattening out a level of composition from a target **CLASS** instances. These restructuring operands are particular and required in the presence of data models capable of building composite structures. These operands have their origin to the introduction of non-first normal form relational models. In our implementation projection is included in the operator's implementation.

12.1.2.1 Nest

The unary nest by value algebraic operator partitions the class instances according to a list of properties, which is passed as an argument, and collects the distinct values of another property into a new set based property. Therefore from every partition of the partitioning properties a set of values of the nesting property is built. While the list of partitioning properties and nesting property has a single cardinality data type requirement, the cardinality of the nested attribute of the output is a set. Consequently the data type signature of a nest by value operand has the same signatures for the partitioning attributes but it changes the name and cardinality (i.e. to set) of the nested property. There is a specific point to our data and query model that needs addressing is whether the extent or deep extent of the arguments is required.

The following example partitions the **CLASS TRAN'S** deep extent by property **YEAR'S** distinct values and nests the **SALE** property. That is for every **YEAR'S** distinct value, all values of **SALE**

coming from objects that have the same **YEAR'** instance value are collated into a set; which is renamed to **NESTSALE**. The third query shows the data type signature of the query instance: while the **YEAR** signature is identical to **TRAN'S**, the property **NESTSALE** has a different name and upper cardinality from the property **SALE** of **TRAN**. The last query shows some of the set element values of **NESTSALE** for different partitions.

```
?- algebra[%nv(tran,"deepextent",[year],sale)].
Nesting arguments exist: ok
method not found list: []
Nesting methods exist: ok
method not found list: []
Attr to nest exist: ok
algquery instance created
signatures created
data loid and aggregate done
Yes.
?- ?Q:algquery.
?Q = nv(tran,"deepextent",555)
Yes.
?- ?Q:algquery[?M{?L:?U}*=>?Dt;?M{?L:?U}=>?Dt].
?Q = nv(tran,"deepextent",555)  ?M = nestsale   ?Dt = integer   ?L = 0  ?U = *
?Q = nv(tran,"deepextent",555)  ?M = year       ?Dt = integer   ?L = 1  ?U = 1
Yes.

?- ?I:nv(tran,"deepextent",555)[?M->?D].
?I = inv(tran,"deepextent",555,2010)   ?M = nestsale   ?D = 100
?I = inv(tran,"deepextent",555,2010)   ?M = nestsale   ?D = 200
…
?I = inv(tran,"deepextent",555,2012)   ?M = nestsale   ?D = 250
?I = inv(tran,"deepextent",555,2012)   ?M = nestsale   ?D = 350
?I = inv(tran,"deepextent",555,2012)   ?M = nestsale   ?D = 450
?I = inv(tran,"deepextent",555,2012)   ?M = nestsale   ?D = 550
?I = inv(tran,"deepextent",555,2012)   ?M = year       ?D = 2012
Yes.
```

For the nest operator implementation two reified rules are used. These are attached to an object identified with **NVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert a **NV(…)** object is an instance-of **ALGQUERY**. The method takes five arguments: the first two are the collections on which the nesting is to be executed and an indicator to state whether a deep extent or an extent is required for the result. The third contains a list of property names to partition on. The forth argument is bound to a class property which is to be nested. The fifth argument is a unique identifier. Note that object **NV(…)** instantiated has two properties that unify with the list of partitioning properties and the property to be nested.

```
nvf   [algqueryinstance(?C1,?Deep,?Usmthlst,?Nestattr,?Tag) ->
  ${  (  nv(?C1,?Deep,?Tag):algquery[nestby->?Usmthlst, nestatr->?Nestattr]
         :- true  ) }].
```

The second reification concerns the building of rules for creating data type signature of the nesting result. The data type signature of the query result is determined by the list of

partitioning properties and the property to be nested. Also required is the creation of a name for the nested property. For example nested property **SALE** is renamed to **NESTSALE**. The data type reification starts by sifting for the relative **ALGQUERY** instance of, i.e. **NV(…)**. For each property in the method list bound to **?USMTHLST** we search for all non-compound inheritable methods and bind it to **?M**. This property data-type signature is instantiated to the query instance. In the case of the nested attribute we need to do two things once the nested property is unified to variable **?M** (this is the second reification instance). The first is renaming, and the second is setting the upper bound from **1** to '*****' (i.e. many). There are eight rules based on the property's mode (e.g. inheritable or not) and whether part of the partitioning set or nesting attribute.

```
nvf[algquerydt(?C1,?Deep,?Usmthlst,?Nestattr,?Tag)  ->
  ${                                                  // nest by attributes
     (  nv(?C1,?Deep,?Tag)[?M{?B:?T}*=>?D]  :-
           nv(?C1,?Deep,?Tag):algquery,
           %lmember(?M,?Usmthlst),
           ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog
      ), … ,
                                                      // nested attribute
     (  nv(?C1,?Deep,?Tag)[ ?Maggr{0:*}*=>?D]  :-
           nv(?C1,?Deep,?Tag):algquery, ?Nestattr=?M,
           ?C1[?M*=>?D], not compound(?M)@_prolog,
            addprefix('nest',?M,?Maggr)
      ), …
    }].
```

The actual evaluation of nesting by value is done through procedure **%NV** in object **ALGEBRA**. The procedure **%NV** takes four arguments: the first two are the classes' instances and an indicator on the extent required (i.e. unified to variable **?DEEP**). The third argument is a list of properties on which to nest. The final argument is bound to the property to be nested. The first part of the implementation deals with pre-checks: for example the partitioning properties list contains at least one property and every property exists and has singular cardinality. Another check is done on the nested attribute – again it must exist and has singular cardinality too. Once the checks are confirmed an identifier is generated and it is unified with **?TAG**. The procedure then creates the query instance and the query instance data-type signature; as usual these are handled by the reification of the rules and inserting them into the object base. The instantiation of the query instance objects is slightly different here: first, but as in aggregate operator, the insert of a fact into the object base is done through the Flora-2 parse module; and second, the fact instantiation is split into two parts (i.e. one deals with partitioning values and the other deals with the nested

property). Two separate procedures are called, namely **%NVLOIDDATA11** and **%NVLOIDDATA12**, in sequence.

```
algebra[%nv(?C1,?Deep,?Usmthlst,?Nestattr)] :-
    ?Nigb =count{?Igb | %lmember(?Igb, ?Usmthlst)   },
    if ( ?Nigb > 0 ) …,
    algebra[%singlemethodsexist (?C1,?Usmthlst,?Nousmthlst)],
    if ( ?Nousmthlst = [] ) …,
    algebra[%singlemethodsexist(?C1,[?Nestattr],?Nonestattr)],
    newoid{?Tag},                                    // get a tag
    nvf[ algqueryinstance(?C1,?Deep,?Usmthlst,?Nestattr,?Tag) -> ?Rules1 ],
    insertrule { ?Rules1 },                          // create query instance
    writeln('algquery instance created')@_prolog,
    nvf[ algquerydt(?C1,?Deep,?Usmthlst,?Nestattr,?Tag)-> ?Rules2 ],
    insertrule { ?Rules2 },
    writeln('signatures created')@_prolog,
    //generate aggregate loid and data from nesting
    if ?Deep="deepextent" then (?Deepc='deepextent') else (?Deepc='extent'),
    %nvloiddata11(?C1,?Deepc,?Usmthlst,?Nestattr,?Tag),
    %nvloiddata12(?C1,?Deepc,?Usmthlst,?Nestattr,?Tag),
    writeln('data loid and aggregate done')@_prolog.
```

The two procedures share a common logic based on incrementally building a string, from the arguments provided, to build a rule. Once this rule is built it is parsed and inserted into the object base for evaluation. The procedure, actually both of them, takes the same arguments as the calling procedure.

```
    %nvloiddata11(?C1,?Deep,?Usmthlst,?_Nestattr,?Tag):-
      sc('"',?Deep,'"',?Deepe),
      ?T0='inv(', ?L1=[?C1,?Deepe,?Tag],l2sc(?L1,?T0,?T1),
      sc(?T1,',',?T2),
      ml2sc(?Usmthlst,?T2,?T3),
      ?T4='):nv(', sc(?T3,?T4,?T5),
      ?L2=[?C1,?Deepe,?Tag],l2sc(?L2,?T5,?T6),
      ?T7=')[',sc(?T6,?T7,?T8),
      mvl2sc(?Usmthlst,?T8,?T9),
      sc(?T9,'] :- ?I:',?C1,' , ?I[',?T10),
      mvl2sc(?Usmthlst,?T10,?T11),
      sc(?T11,'].',?T12),
      %insertrulebyparseandeval(?T12).
```

In the following listing the rules generated for a query are given:

```
Remark    Example query
Remark    algebra[%nv(tran,"deepextent",[year],sale)].

Remark    Two rules generated for nesting sale by year

inv(tran,"deepextent",…,?year):nv(tran,"deepextent",…)[year -> ?year]
   :-    ?I:tran , ?I[year -> ?year].

inv(tran,"deepextent",…,?year)[nestsale -> ?sale]
   :-    ?I:tran , ?I[year -> ?year, sale -> ?sale].
```

12.1.2.2 Unnest

The unary unnest operator unfolds a set valued property into single valued property; each element of the set of values is 'joined' to a set of values pertaining to list of properties passed as arguments to the operator. It needs to be stated that while the unnesting property has set cardinality the other properties have single cardinality. In simple terms it undoes what nest

operator does; but more of this later. Consequently the data-type signature of an unnest by value operand has the same signatures for the repeating attributes but it changes the name and cardinality (i.e. to single cardinality) of the unnested property. There is a specific point for our data and query model that needs addressing as to whether the extent or deep extent of the arguments is required. Also a duplicate post evaluation check needs to be run and any duplicates eliminated.

The following script shows an unnesting example. The first query actually show two attribute values of object identified as **MARY**: **FNAME** is a singular property and takes a string **"MARY"**, while **TELNO** is set valued property and it takes two numbers **1234** and **5678**. The second query fires the unnest operator on set value property **TELNO** and with property **FNAME** as the repeating value in output object. The verbose output lists the evaluation progress including comments from the pre and post checks. The next query extracts the data type signature of the query instance created for the unnesting by value operation. Note the renaming of the unnested attribute – i.e. **UNTELNO**. The next query shows the properties of the query instance: i.e. the query arguments. The result of the operation is shown in the last query (i.e. sanitised output); it has to be noted that there two instances of **OV(…)** that hold the string **"MARY"**, i.e. repeating property, and the telephone number **1234** and **5678** respectively. Indication that these are two separate instances is given in the functor logical identifier of relative instances – e.g. **IOV(…)**.

```
?- mary[fname->?F, telno->?T].
?F = "mary"^^_string   ?T = "1234"^^_integer
?F = "mary"^^_string   ?T = "5678"^^_integer
Yes.
?- algebra[%ov(student,"deepextent",[fname],telno)].
Unnesting arguments exist: ok
method not found list: []
Unnesting methods exist: ok
method not found list: []
Attr. to unnest exist: ok
algquery instance created
signatures created
data loid and unnesting done
duplicate list[]
duplicate deletion done
Yes.
?- ?Q:algquery[?M*=>?Dt;?M=>?Dt].
?Q = ov(student,"deepextent",…)    ?M = fname       ?Dt = string
?Q = ov(student,"deepextent",159) ?M = untelno     ?Dt = integer
Yes.
?- ?Q:algquery[?M->?V].
?Q = ov(student,"deepextent",159)   ?M = unnestatr   ?V = telno
?Q = ov(student,"deepextent",159)   ?M = unnestby    ?V = [fname]
Yes.
```

```
?- ?I:ov(…)[?M->?V].
?I = iov(…,"mary"^^_s,"1234"^^_i)     ?M = fname    ?V = "mary"^^_s
?I = iov(…,"mary"^^_s,"1234"^^_i)     ?M = untelno  ?V = "1234"^^_i
?I = iov(…,"mary"^^_s,"5678"^^_i)     ?M = fname    ?V = "mary"^^_s
?I = iov(…,"mary"^^_s,"5678"^^_i)     ?M = untelno  ?V = "5678"^^_i
?I = iov(…,"michael"^^_s,"3456"^^_i) ?M = fname    ?V = "michael"^^_s
?I = iov(…,"michael"^^_s,"3456"^^_i) ?M = untelno  ?V = "3456"^^_i
?I = iov(…,"michael"^^_s,"9012"^^_i) ?M = fname    ?V = "michael"^^_s
?I = iov(…,"michael"^^_s,"9012"^^_i) ?M = untelno  ?V = "9012"^^_i
…
Yes.
```

For the unnest operator implementation two reified rules are used. These are attached to an object identified with **OVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert an **OV(…)** object is an instance-of **ALGQUERY**. The method takes five arguments: the first two are the collections on which the mapping is to be executed and an indicator to state whether a deep extent or an extent is required for the result. The third contains a list of property names to repeat on. The forth argument is bound to class property which is to be unnested. The fifth argument is a unique identifier. Note that object **OV(…)** instantiated has two properties that unify with the list of repeating properties and the property to be unnested.

```
ovf[ algqueryinstance(?C1,?Deep,?Usmthlst,?Unnestattr,?Tag) ->
  ${(  ov(?C1,?Deep,?Tag):algquery [nestby->?Usmthlst, nestatr->?Unnestattr]
         :- true  )  }].
```

The second reification concerns the building of rules for creating the data-type signature of the unnesting result. The data type signature of the query result is determined by the list of repeating properties and the property to be unnested. Also required is the renaming of the unnested property. For example unnested property **TELNO** is renamed to **UNTELNO**. The data type reification starts by sifting for the relative **ALGQUERY** instance of, i.e. **OV(…)**. For each property in the method list bound to **?USMTHLST** we search for all non-compound inheritable methods and bind each to **?M**. This property data-type signature is instantiated to the query instance. In the case of the unnesting attribute we need to do two things once the unnested property is unified to variable **?M** (this is the second reification instance). The first is renaming, and the second is setting the upper bound from '*****' to **1** (i.e. one). There are, as usual, four rules based on the property's mode (e.g. inheritable or not) for repeating properties and four for the unnesting attribute.

```
ovf[ algquerydt(?C1,?Deep,?Usmthlst,?Unnestattr,?Tag)  ->
  ${                                                  // unnest by attributes
      (  ov(?C1,?Deep,?Tag)[?M{?B:?T}*=>?D]  :-
            ov(?C1,?Deep,?Tag):algquery,
            %lmember(?M,?Usmthlst),
            ?C1[?M{?B:?T}*=>?D],
            not compound(?M)@_prolog ), …

      (  ov(?C1,?Deep,?Tag)[ ?Maggr{0:*}*=>?D]     :-
            ov(?C1,?Deep,?Tag):algquery,
            ?Unnestattr=?M,
            ?C1[?M=>?D],
            not compound(?M)@_prolog,
            addprefix('un',?M,?Maggr) …
   }  ].
```

The actual evaluation of unnesting by value is done through procedure **%OV** in object **ALGEBRA**. The procedure **%OV** takes four arguments: the first two are the classes' instances and an indicator on the extent required (i.e. unified to variable **?DEEP**). The third argument is a list of properties on which to repeat the unnesting property. The fourth and final argument is bound to the property to be unnested. The first part of the implementation deals with pre-checks: for example the repeating properties list contains at least one property and every property exists and has singular cardinality. Another check is done on the unnesting attribute through method **%SETMETHODSEXIST** of object **ALGEBRA** – again it must exist and has set cardinality too. Once the checks are confirmed an identifier is generated; it is unified with **?TAG**. The procedure than creates the query instance and the query instance data type signature; these are handled by the reification of the rules and inserting them into the object base. The instantiation of the query instance objects is slightly different of the nest operator: the rule to generate the instances is constructed into a string and then the string is parsed and evaluated into an object base through the parse module. Procedure is **%OVLOIDDATA** called. A post check for duplicates is made.

```
algebra[%ov(?C1,?Deep,?Usmthlst,?Unnestattr)] :-
  ?Nigb =count{?Igb | %lmember(?Igb, ?Usmthlst)   },
  if ( ?Nigb > 0 ) … ,
  algebra[%singlemethodsexist (?C1,?Usmthlst,?Nousmthlst)],
  if ( ?Nousmthlst = [] ) … ,
  algebra[%setmethodsexist(?C1,[?Unnestattr],?Nounnestattr)],
  if ( ?Nounnestattr = [] ) … ,
  newoid{?Tag},
  ovf[ algqueryinstance(?C1,?Deep,?Usmthlst,?Unnestattr,?Tag) -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  ovf[ algquerydt(?C1,?Deep,?Usmthlst,?Unnestattr,?Tag)-> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  // generate aggregate loid and data from aggregation
  if ?Deep="deepextent" then (?Deepc='deepextent') else (?Deepc='extent'),
  %ovloiddata(?C1,?Deepc,?Usmthlst,?Unnestattr,?Tag),
  writeln('data loid and unnesting done')@_prolog,
  algebra[%duplicates( ov(?C1,?Deep,?Tag),?Duplst)],
  write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
  algebra[%delduplicate(?Duplst, ov(?C1,?Deep,?Tag))].
```

The calling of procedure **%OVLOIDDATA** produces a string which is a rule: for this example the rule says if a student instance-of whose properties **FNAME** and **TELNO** values are unified to variables **?FNAME** and **?TELNO** exist then create an **IOV(…)** object, that is an instance-of **OV(…)**, and whose properties **FNAME** and **UNTELNO** are unified to variables **?FNAME** and **?TELNO**. Note how the rule takes care of building the identifier, especially that of **IOV(…)**, by including the values of two variables **?FNAME** and **?TELNO** in its functor.

```
Remark    Example query
Remark    algebra[%ov(student,"deepextent",[fname],telno)].

Remark    A  rule generated for unnesting of telno and repeating year

iov(student,"deepextent",159,?fname,?telno):ov(student,"deepextent",159)
[fname->?fname, untelno->?telno] :-
          ?I:student, ?I[fname->?fname,telno->?telno].
```

12.1.2.3 Properties of Nest and Unnest operators

The application of a nesting to an unnesting and the application of unnesting to nest have well known properties; these are attributed to many sources but are mainly credited to Roth, Korth & Silberschatz in [ROTHM88], and Thomas and Fischer in [THOMA86].

In general unnest (denoted byμ) is the inverse of nest (denoted by υ):

$$\mu\left(\upsilon\left(class\right)\right) = class.$$

In general it is not the case that nest is the inverse of unnest:

$$\upsilon\left(\mu\left(class\right)\right) \neq class.$$

('class' denotes ranges – e.g. **CLASS** instance).

For nesting to be the inverse of unnest the grouping properties must functionally determine the nesting property.

12.1.3 Rename Operator

The unary rename operator changes the names of a class instance properties. The rename operator, for example, can get two non-union compatible arguments compatible, if only their property names are different. In the following script we show how to "join" together the names of units and student (i.e. **UNAME** and **FNAME** respectively) in one query instance output. In the first two queries each expression projects out the relative properties of **UNAME** and **FNAME**. Then a union by value is attempted but fails as the input arguments are not union compatible because the property name does not match. The rename operator is called twice and renames each

attribute, of the projected output query instance results, into **RNAME**. At this point the union by values is executed and succeeds. Some of the last query's output is included.

```
?- algebra[%pv(student,"extent",[fname])].
Yes.
?- algebra[%pv(unit,"extent",[uname])].
Yes.
?- ?Q:algquery, ?I:?Q.
?Q = pv(student,"extent",_#'18219)   ?I = ipv(student,"extent",_#'18219,mary)
?Q = pv(student,"extent",_#'18219)   ?I = ipv(student,"extent",_#'18219,michael)
?Q = pv(student,"extent",_#'18219)   ?I = ipv(student,"extent",_#'18219,nancy)
…
?Q = pv(unit,"extent",_#'18220)      ?I = ipv(unit,"extent",_#'18220,fyp)
Yes.
?- ?Rt=pv(unit,"extent",?_1),
   ?Lf=pv(student,"extent",?_2),
   algebra[%uv(?Lf,?Rt,"extent")].
not union compatible
No.
?- ?Rt=pv(unit,"extent",?_1), algebra[%rv(?Rt,"extent",[uname],[rname])].
Yes.
?- ?Lf=pv(student,"extent",?_2), algebra[%rv(?Lf,"extent",[fname],[rname])].
Yes.
?- ?Lf:algquery, ?Lf=rv(pv(student,"extent",?_1),"extent",?_2),
   ?Rt:algquery, ?Rt=rv(pv(unit,"extent",?_3),"extent",?_4),
   algebra[%uv(?Lf,?Rt,"extent")].
union compatible
UC: ok
…
Yes.
?- ?_Q:algquery, ?_Q=uv(?_1,?_2,?_3), ?_I:?_Q, ?_I[rname->?N].
?N = "data struct & algo"^^_string
?N = "dist db"^^_string
…
?N = "robert"^^_string
?N = "soft eng"^^_string
Yes.
```

For the rename operator implementation two reified rules are used. These are attached to an object identified with **RVF**.

The property **ALGQUERYINSTANCE** takes care to have a parameterised version and thus assert a **RV(…)** object is an instance-of **ALGQUERY**. The method takes five arguments: the first one is a collection on which the rename is to be executed; e.g. a class name. The second contains an indicator to state whether a deep extent or an extent is required for the result. The third and fourth contains lists of methods to rename (i.e. from old to new names – henceforth called 'from' and 'to' lists). The fifth is an auto-generated logical identifier. Other than creating an instance-of, the reification of the rule includes instantiation of two properties: these are assigned the 'from' and the 'to' lists (i.e. in **RENAMEFROM** and **RENAMETO** in **RV(…)**).

```
rvf  [algqueryinstance(?C1,?Deep,?Fmthlst,?Tmthlst,?Tag) ->
  ${ (rv(?C1,?Deep,?Tag):algquery [renamefrom->?Fmthlst, renameto->?Tmthlst]
        :- true  ) }].
```

It has been just stated that 'from' and 'to' lists are two arguments that govern the renaming operation. A simple predicate is required to map a property name in the 'from' list to the

corresponding name in the 'to' list. The predicate is called **%LOOKUP212** and takes four arguments: the first two are the 'from' and 'to' lists, the third is a name in the 'from' list and the forth being the item in the 'to' list that comes in the same sequence as the third argument. The following shows how the procedure unifies '**C**' from the 'to' list for the item '**3**' in the 'from' list.

```
?- ?Source=[1,2,3,4],?Dest=[a,b,c,d],%lookup121(?Source,?Dest,3,?M).
?Source = [1, 2, 3, 4]    ?Dest = [a, b, c, d]    ?M = c
Yes.
```

The second reification concerns the data type signature of the result. Actually the data type signature remains intact except for the renaming of some property's name. The **RVF** object uses the **ALGQUERYDT** property and it takes five parameters: the same set as **ALGQUERYINSTANCE**. The **ALGQUERYDT** has four instantiations each related to a different properties mode–cater for inheritable (i.e. ***=>**) and non-inheritable (i.e. **=>**) methods, and arity and non-arity (i.e. **NOT COMPOUND(?M)**) methods. Each rule has the same pattern but each fire for a particular data signature type. If there is a method signature in an argument class then it needs to be asserted in the output data type signature (i.e. of **RV(…)** object). What is particular here is the part where if a property being evaluated, i.e. **?M**, is a member of the 'from' list, i.e. **?FMTHLST**, then a look up to its corresponding 'to' list entry is made through predicate **%LOOKUP121**.

```
rvf  [algquerydt(?C1,?Deep,?Fmthlst,?Tmthlst,?Tag)  ->
  ${ (  rv(?C1,?Deep,?Tag)[?Rm{?B:?T}*=>?D]   :-
          rv(?C1,?Deep,?Tag):algquery,
          ?C1[?M{?B:?T}*=>?D], not compound(?M)@_prolog,
          if   ( %lmember(?M,?Fmthlst) )
          then ( %lookup121(?Fmthlst,?Tmthlst,?M,?Rm) )
          else ( ?M=?Rm ) ),
      … ].
```

The object **ALGEBRA** has procedure **%RV** to implement rename by value; also the procedure takes four arguments. The actual implementation of rename operation is long; this is due to a number of preconditions to be tested for proper application of the 'from' and 'to' renaming – these are the third and fourth arguments of **%RV**. The first test is whether the two lists have the same number of elements. The second test ensures that all elements in the 'from' list are actually methods found in the class instance, this is the first argument of procedure. The method used is the same one used in the project by value operator: **%METHODEXIST** of object algebra. The third check ensures that no name in the 'to' list is already found in the class instance – again using procedure %**METHODSEXIST**. The fourth check, and final pre-check, needs to check that if a property to be renamed is compound then only the name is changed; i.e. the argument data type is not. An

appropriate procedure, called **%RVNOMTHPARAMCHANGE** and explained later, is invoked with the 'from' and 'to' lists. It fails if there is such an issue. After the pre-checks the **%RV** procedure generates a unique logical identifier; which is unified to **?TAG**. The two reified rules are invoked, reified, and inserted into the object base – these take care of instance-of assertion (i.e. **IRV(…):RV(…)**) and data type signatures of query instance class. As is common across most operators, the next actions are to call procedures **%RVDATALOID** and **%RVDATA** so as to create the query extent and transfer data onto the query instance collection. The operator ends by checking for duplicates and purges any present.

```
algebra[%rv(?C1,?Deep,?Usfmthlst,?Ustmthlst)] :-
  ?Nfmth=count{ ?Ifmth | %lmember(?Ifmth, ?Usfmthlst) },
  ?Ntmth=count{ ?Itmth | %lmember(?Itmth, ?Ustmthlst) },
  if    ( ?Nfmth = ?Ntmth )
  then  ( writeln('From and To Methods list number of elements: ok')@_p)
  else  ( writeln('From and To Methods list have diff number of ele.')@_p,
          fail ),
  algebra[%methodsexist (?C1,?Usfmthlst,?Nofmthlst)],
  if    ( ?Nofmthlst = [] )
  then  ( writeln('From Methods exist: ok')@_prolog)
  else  ( write('From Methods list: problem; missing attributes - ')@_prolog,
          writeln(?Nofmthlst)@_prolog,
          fail ),
  algebra[%methodsexist(?C1,?Ustmthlst,?Notmthlst)],
  if    ( ?Notmthlst = ?Ustmthlst )
  then  ( writeln('To Methods have no clash: ok')@_prolog)
  else  ( write('To Methods list: problem clashing attributes - ')@_prolog,
          writeln(?Ustmthlst)@_prolog,
          fail),
  %rvnomthparamchange(?Usfmthlst,?Ustmthlst),
  newoid{?Tag},
  rvf[ algqueryinstance(?C1,?Deep,?Usfmthlst,?Ustmthlst,?Tag) -> ?Rules1 ],
  insertrule { ?Rules1 },
  writeln('algquery instance created')@_prolog,
  rvf[ algquerydt(?C1,?Deep,?Usfmthlst,?Ustmthlst,?Tag)-> ?Rules2 ],
  insertrule { ?Rules2 },
  writeln('signatures created')@_prolog,
  %rvdataloid(?C1,?Deep?Usfmthlst,?Ustmthlst,?Tag),
  %rvdata(?C1,?Deep,?Usfmthlst,?Ustmthlst,?Tag),
  writeln('data loid and move done')@_prolog,
  algebra[%duplicates(rv(?C1,?Deep,?Tag),?Duplst)],
  write('duplicate list')@_prolog, writeln(?Duplst)@_prolog,
  algebra[%delduplicate(?Duplst,rv(?C1,?Deep,?Tag))],
  writeln('duplicate deletion done')@_prolog.
```

The procedure **%RVDATALOID**, called from **%RV** builds a list of objects that need to instantiated as instances of **RV(…)** – these are instances-of **?C1**. The list is passed to recursive procedure **%RVINSERTLEFT** that iteratively inserts clauses of the form **IRV(…):RV(…)** into the object base.

```
%rvdataloid(?C1,?Deep,?Fmthlst,?Tmthlst,?Tag) :-
  if    (?Deep="extent")
  then (?Ileftlst=collectset{?Ileft|?Ileft:?C1,
                                    if (?Sc::?C1, ?Ileft:?Sc)
                                    then (false) else (true)})
  else (?Ileftlst=collectset{?Ileft|?Ileft:?C1}),
  %rvinsertleft(?Ileftlst,?C1,?Deep,?Fmthlst,?Tmthlst,?Tag).
```

The **%RVDATA** although similar to other operators' implementation, like four sections to deal with each type of method signature, has some differences. The basic difference is that if the method whose value is being copied into is in the 'to' list then the corresponding 'from' list method is sought and data copied from class instance **?C1**. Once the collection is built it is passed on to procedure **%RVINSERTMETHOD** (or **%RVINSERTPMETHOD** for arity properties) for it to assert the property's values.

```
%rvdata(?C1,?Deep,?Fmthlst,?Tmthlst,?Tag) :-
  ?LeftIsmlst =
    collectset{ ?Wl | rv(?C1,?Deep,?Tag):algquery,
                      rv(?C1,?Deep,?Tag)[?M*=>?D],
                      not compound(?M)@_prolog,
                      irv(?C1,?Deep,?Tag,?I)[],
                      if   ( %lmember(?M,?Tmthlst) )
                      then ( %lookup121(?Fmthlst,?Tmthlst,?Oldm,?M),
                             ?I:?C1[?Oldm->?R],dt(?D,?Dt),?R:?Dt,
                             ?Wl=f(?C1,?Deep,?Tag,?I,?M,?R) )
                      else ( ?I:?C1[?M->?R], dt(?D,?Dt), ?R:?Dt,
                             ?Wl=f(?C1,?Deep,?Tag,?I,?M,?R) ) },
  %rvinsertsmethod(?LeftIsmlst),
  … .

  %rvinsertsmethod([?H|?Rlst]):-
    ?H=f(?C1,?Deep,?Tag,?I,?M,?R),
    insert{ irv(?C1,?Deep,?Tag,?I)[?M->?R] },
    %rvinsertsmethod(?Rlst).
  %rvinsertsmethod([]).
```

The procedure **%RVNOMTHPARAMCHANGE** is called with two list unified to it; i.e. 'from' and 'to' lists. The procedure recurs over every item of lists, actually one each at a time, and in each case does three checks. The first check, if both items are compound – i.e. methods with an argument, entails ensuring that both arguments are identical and if it's not the case then the procedure fails. The second and third checks if one is a compound term whilst the other is not and if it is the case then procedure fails.

```
%rvnomthparamchange([?Fmth|?Fmthlst],[?Tmth|?Tmthlst]) :-
  if   ( compound(?Fmth)@_prolog,
         compound(?Tmth)@_prolog,
         ?Tmth=?_T(?Pt),
         ?Fmth=?_F(?Pf) )
  then ( if ( ?Pt!=?Pf )
         then ( wf('method's parameters dont match in from and to lst.'),
                fail ) )
  else ( %rvnomthparamchange(?Fmthlst,?Tmthlst) ),
  if   ( compound(?Fmth)@_prolog,
         not compound(?Tmth)@_prolog )
  then ( wf('method's parameters dont match in from and to lists.'),
         fail )
  else ( %rvnomthparamchange(?Fmthlst,?Tmthlst) ),
  if   ( not compound(?Fmth)@_prolog, compound(?Tmth)@_prolog )
  then ( wf('method's parameters dont match in from and to lists.'),
         fail )
  else ( %rvnomthparamchange(?Fmthlst,?Tmthlst) ).
%rvnomthparamchange([],[]) :- !.
```

Properties of the rename operator

The operator is lossy as a duplicate scan and removal is specified. Also a sequence of renames, on the same property, is equivalent to the last rename.

12.2 Cross Algebraic Operators Properties

The algebraic operators have properties across other operands. (Many advanced database text books have these – e.g. in Silberschatz *et al.* [SILBE10] and they are applicable here). For example select operand can be combined with product and theta joins (i.e. joins not based on equality but any other comparison). Conversely a select predicate made out of a number of conjunctions can be disengaged.

$$\sigma_{\theta}(\mathrm{E}_1 \times \mathrm{E}_2) \Leftrightarrow \mathrm{E}_1 \bowtie_{\theta} \mathrm{E}_2$$

and

$$\sigma_{\theta_1}(\mathrm{E}_1 \bowtie_{\theta_2} \mathrm{E}_2) \Leftrightarrow \mathrm{E}_1 \bowtie_{\theta_1 \wedge \theta_2} \mathrm{E}_2$$

The following are some of the more useful transformations:

- Natural joins are associative;
- Theta joins are also associative (under certain structural limitations);
- Selection operand is distributive over theta joins;
- Projection operand is distributive over theta joins.

As we have previously stated there are a number of techniques that the framework adopts to implement these: the first is reification and instantiation of rules; the second is the technique of equality over logical identifiers; and the third is inserting rules into the object base through Flora-2 parsing module. Furthermore some transformations are better left for later parts of query optimisation because, for example, the numbers of permutations generated are prohibitive and therefore pruning the search space while transforming is inevitable.

12.3 Summary

This chapter built a query model based on an algebra specially developed for our object-database framework. The query model is procedural, closed, typed, and at least Codd complete. The algebra has ten operators which are relational and nested relational in origin, except for one which has a functional origin.

Every operator implemented is sensitive to the underlying object-oriented data model. Each operator takes a collection of objects as input; in our framework this is a class, structure or a query instance. Furthermore each algebraic operator makes extensive use of the underlying meta data available in the framework; e.g. to work out the data type of the result. It needs to be emphasised that operators are value based and yet able to work with logical identifiers.

Each operator has its algebraic properties; these include commutativity, associativity, and distributivity. Through identity equality most of these are implemented in our framework. Furthermore these operators, when composed, derive other known algebraic operators; e.g. intersection and join.

The output of the algebraic expression is an object collection and its objects are made an instance-of an object that represents the query invocation; i.e. the query instance itself is an instance of **ALGQUERY** object. In the query instance complete details of the original query are kept. As stated earlier the query instance collections are then available as operands to other query expressions. (Of course a query instance extent is valid until the range to which it was applied to ceases to remain current).

The implementation of all algebraic operands is in Flora-2. Many important parts of an operator's implementation are declarative; for example working out the data type of the result, and working out the extent. A good number of methods supplement the algebraic operations. Flora-2 features, some of which are advanced, have been used extensively. These features include identity with functors, identity equality, rule reifications, rule parsing from generated strings, inserting and retracting facts from the object base. Some underlying XSB Prolog predicates have been used too. Other methods developed in the object-oriented data modelling are used here too; for example data-type inference and checking methods.

As regards comparison of this algebra to relational algebra in terms of what queries can be specified we assert that ours is a super set; clearly there are no equivalents of aggregate and map. Nonetheless our algebra is not a complete programming language. Notably missing operand in our algebra is the fix-point for recursion.

A possible drawback here could be the shallowness of most operands over the object structures (two operations are an exception – i.e. select and map which both accept path expressions that

navigate deeply into structures); this is actually a consequence of our data model's object constructor choice. For example, when we project on a class extent, only the properties visible at the level of that class are seen. If a property is a composition of another class, or structure, then only its identifier is projected out. There are two contrasting issues. On one side, if operators are shallow they are easy to implement, unambiguous, and each operator's scope does not overlap with another. On the other side, for example in other algebras, some non-shallow projections replace a query based on select, product, and project.

A drawback of our algebra is that it does not handle multi-dimensional arrays and multi-sets (e.g. bags). Also each set or collection is typed: i.e. no collection is heterogeneous in terms of instance-of. There is no reason why the framework, algebra and Flora-2 system cannot offer support for multi-dimensional arrays and multi-sets other than time. A similar problem is the implementation of the operand's result properties: while the procedures to imply associativity and commutativity have been implemented in many cases, in some cases these haven't been implemented. A justification, for most cases, was given in terms of the number of permutations possible. Another limitation, inherited from data-type checking and inference module, is our consideration of methods with one argument. It is possible to include methods with any number of arguments but our coding needs a different approach; nonetheless it is doable.

A qualitative argument that many do well to question about programming languages is their readability and writeability. We do not think it is overly complex to read the algebraic query constructs and results; but it is acceptable when a criticism is levelled at identifier's more involved structures (e.g. identifiers with involved functors). Clearly a decent integrated development environment with folding and unpacking over the framework's object base is required.

We have many times shown a query composed from other queries. We have implied that each query has been materialised – i.e. asserted as instances of the query instance. It must be stated that none of the query instances maintain its currency after evaluation; i.e. it is assumed that the object base, especially the parts the query instances have been computed from has not changed. If changes are made a re-evaluation of the queries is required if still in demand. As we shall see

in the next chapters the assumption that an object bases state does not change, except from query processing is not prohibitive or pointless in some circumstances.

The framework and algebraic operators have been implemented with Flora-2. Flora-2 implements logic programming with F-logic and the latter's distinctive higher-order syntax, first order semantics, and object-oriented semantics ingrained in its inference. Flora-2 advanced features, like rule reification, parsing strings for rules and evaluating them in the object base, equality of logical identifiers, have contributed immensely to the implementation of the algebraic operators. Many of the implementations have high level and declarative constructs. Nonetheless it is reasonable to find parts of the code base that are procedural and in need of a code refactoring exercises.

This chapter and the previous has the opportunity to merge a number of trends developed in this study: firstly, the integration of a conceptual design encoding into the framework, development of type checking and type inference, and finally query language constructs that work over the framework. The next chapter shows how the algebra is used as a procedural target of a real declarative object query language and shows that the design and query model can aid query processing and optimisation.

Chapter 13

Query Processing and Optimisation

13 Query Processing and Optimisation

In this chapter a qualitative structural translation for a subset of ODMG OQL constructs to the algebra developed earlier is illustrated. The subset includes: constructs whose output constructor is a set; constructs that restructure the output through packing and unpacking of object database structures; constructs that include a sequence of target sets (of objects); a selection of filter constructs; and group by queries.

There are a number of ODMG OQL features that are not implemented. These include bag structure for output results, some filter conditions, the having clause in the group by; and the sorting clause. A special section for these shortcomings had to be surveyed. This is a long list and it is mainly due to two causes: the first is time required to implement; and the second is the fact that ODMG query model does have an extensive list of features.

The latter half of this chapter shows how algebraic expressions, possibly encoded expressions of OQL queries, are optimised. Various optimisation techniques are presented; most of which are heavily reliant on the underlying object-oriented data model and framework facilities. These techniques include join reductions, semantic optimisation, view materialisation, and avoiding running the duplicate detection and elimination process from query execution.

13.1 ODMG OQL and our Algebra translation

The OQL construct is based on the "select from where" structure popular in SQL. With our algebra a canonical translation of this is a composition of a sequence of product, with possible renaming, selection, and finally a project.

The considerations and translations follow the sequence of ODMG OQL standard's text (i.e. chapter four of [CATTE90]). Each section indicates the key features missing in our algebra and shallow indications on how one can address these. Also we have named the following sub-sections as they are named in the standard's chapter four.

13.1.1 Query Input and Result (4.3)

The main OQL input range is a class. In our algebra the input ranges are supported by **`CLASS`**, **`STRUCTURE`**, and **`ALGQUERY`** instances. The output from the following OQL is exactly what output structure our algebra provides:

```
SELECT DISTINCT STRUCT(a: x.age, s: x.sex)
  FROM Persons x
 WHERE x.name="Pat";
```

The algebra query would be a select on **CLASS** instance **PERSONS** followed by a rename of properties **AGE** and **SEX** into **A** and **S** respectively.

In the following table 13.1 are items mentioned in the relative OQL section but not implemented.

OQL Feature	Severity	Possible resolution	Implementation effort
Concept of a named object	Low	Identifier equality in F-logic between object identifier and name, and check against homonyms	Low
Output of a multi-set	High	Introduce multi-set data type constructor and their semantics. Also have conversion from to and from sets.	Very High
Output not a set of structure	Medium	Have to introduce the notion of values that are distinct from objects.	Very high

Table 13.1: Query input and result

13.1.2 Dealing with Object Identity (4.4)

Like the standard our algebraic operators can access by identifier or by property value.

This section deals with creating objects through query processing; if their nature is transient rather than persistent their query instances of the algebra suffice. Also the OQL standard gives examples of how to access objects in the object database: by property equal to a literal; by object's logical identifier; by projecting an object's property value. All of these are supported by our algebra.

In the following table 13.2 is an item mentioned in relative OQL section but not implemented.

OQL Feature	Severity	Possible resolution	Implementation effort
Creating objects	Low	Assert persistent objects, if not affecting current query, and have to re-evaluate object base; e.g. clear any query instances and their extent.	Low

Table 13.2: Dealing with object identity

13.1.3 Path Expressions (4.5)

In ODMG OQL path expressions must traverse singular properties. In our algebra two operators depend heavily on path expressions: namely select and map. For example in the map operand section we indicated that the complete data type signature, i.e. including cardinality, is computed for any path expression. It is anticipated that if the OQL select list has path expressions, a very common design pattern, then the algebraic map operator is used rather than the algebraic project.

The following table 13.3 is an item mentioned in relative OQL section but not implemented in our algebra:

OQL Feature	**Severity**	**Possible resolution**	**Implementation effort**
andthen and orelse	Low	These Boolean operators are somehow not in loop with query rewriting as their use, for example, drops the commutativity properties of Boolean conjunction. Consequently one may expect their usefulness in 'hand coded' programs that access the object base.	High

Table 13.3: Path expressions

13.1.4 Undefined Values (4.6)

The standard has a specially defined literal called **`UNDEFINED`**. Also a predicate called **`IS_DEFINED(…)`** is explained. In our algebra there is a similar predicate and it is used in the algebraic select operator's predicate. In our algebraic map operator we expect that the user defined value tackles null values. For example Oracle and PostgreSQL have functions that replace a null with an expression's result; these are called **`NVL()`** and **`COALESCE()`** respectively.

In the following table 13.4 are values mentioned in relative OQL section but not implemented.

OQL Feature	**Severity**	**Possible resolution**	**Implementation effort**
UNDEFINED literal for all basic domains.	Medium	Need to revisit and rewrite the data-typing rules of our object collection.	High
Nulls in aggregate functions	Low	Redesign and or replace Flora-2 aggregate primitives.	High

Table 13.4: Undefined values

13.1.5 Method Invoking (4.7)

OQL expects to call a property, i.e. a method, with or without arguments. In our data model and query model properties with an argument are supported too. But note that only one argument is supported. We can state that the limitation is really with coding efficiency rather than impossibility.

13.1.6 Polymorphism (4.8)

The standard asserts that OQL is a typed query language. Also it is expected that polymorphic collections are indicated as ranges to query expressions.

In our framework the data-type checking and inference supports polymorphism and the object collection state is checked for type errors. Each operator uses type checking not only to type the output but also when evaluating the operand's semantics.

In this section reference is made to the extent and deep extent of a query range. In OQL the default is deep extent. Furthermore the standard introduces a predicate that checks, at runtime, the instance-of relationship within a deep extent. – called the *class indicator*. For example **STUDENT** *ISA* **PERSON**, and if one sets the range of a query to be **PERSON'S** deep extent one can then check the instantiating class of each object. In our algebra, specifically, in the select operator, there is a predicate that checks for the instantiating class. It is also possible to have it in a map function too. But there is a difference in our algebra: the methods in the scope of the extent or deep extent are identical; i.e. **STUDENT'S** own methods, i.e. not inherited, are not visible by the range. If this is required then one can go around this issue by rewriting the query. The use of our algebraic operators union and difference is essential and useful.

13.1.7 Operator Composition (4.9)

The standard claims that OQL is a purely functional language. It introduces the construct **DEFINE** to name a query. Defined names can then form part of another query's range.

In our framework algebraic queries are all automatically named and made an instance of object **ALGQUERY**.

13.1.8 Language Definition (4.10)

The main purpose of the section is to ensure that any expression, i.e. query, in OQL is type checked and the output's data type is worked out. OQL uses many type constructors: set, bag, list and array. The section starts by defining type compatibility. It continues by giving typing rules for queries, named query definition, and atomic literals. The next section is a substantial part and deals with data typing of constructor artefacts (e.g. constructing sets, bags, lists). The next section deals with atomic expressions (e.g. binary, string, object identity expressions). There is a section devoted to universal and existential quantification over collections; this is followed by aggregate functions.

In our framework data type and inference there are similar rules but there is no support for bag and array type constructors (as already indicated earlier). In our algebra there is no provision to construct artefacts other than those created through query processing. Ideally this is rectified; an indication of this solution is given in other algebras where a tuple constructor addresses some of these shortcomings. As for aggregate the type of functions available match.

13.1.9 Select Expression–Language Definition (4.10.9)

In OQL the general structure of a retrieval query follows:

```
select [distinct] f(x1, x2,...,xn, xn+1, xn+2,...,xn+p)
  from x1 in e1(xn+1, xn+2,...,xn+p)
       x2 in e2(x1, xn+1, xn+2,...,xn+p)
       x3 in e3(x1, x2, xn+1, xn+2,...,xn+p)
       …
       xn in en(x1, x2,...,xn-1, xn+1, xn+2,...,xn+p)
[where p(x1, x2,...,xn, xn+1, xn+2,...,xn+p)]
```

Each expression, i.e. e_i, in the form clause is a collection or set type. The predicate **`P(…)`** in the where clause is a Boolean expression. The data type of the output is determined by the return type of **`F(..)`**; if return type is *struct(t)* then it is a collection of *struct(t)*.

To compute the above OQL expression in our algebra there are three canonical steps. Some steps have alternatives and these are identified by query's design pattern.

The first does a binary product through the from clause entries (i.e. e_1 to e_n) to generate a collection of objects. There are alternatives to running a sequence of products: one option deals with singular path expressions in select operands; and another deals with set path expressions and using the unnest algebraic operator. For the first alternative to match a subset of expressions e_1 to e_n must convert into a singular path expression (i.e. singular because OQL

recognises only single result path expressions). For example if e_1, e_2, and e_3 connect into a singular path expression then their product is replaced by a single expression with a select based on the path expression. For the second alternative to become effective a subset of e_1 to e_n must convert into a set path expression. It is recommended that for these cases the product operator present is replaced by unnesting.

The second is to sift this collection for objects that fit the query pattern, the Boolean predicate **P(…)** is applied to every object found in the first step and only those that satisfy it are on the output. If deep extent is indicated in the query expression then certain operations between class instances related through the ISA relationships require the use of union and difference operands.

In the third step every object for output is passed to function **F(…)**. If **DISTINCT** keyword is indicated, and we expect so because of our query model, then output has to have duplicates purged and the data-type constructor is formally a set rather then a collection. The structure of function **F(…)** is realised by a combination of project, map, nest, and unnest operands.

Therefore the canonical translation, or better still the naïve translation, of the OQL structure above with **DISTINCT** mandatory follows. First the product operator is repeatedly called to create the Cartesian product of all expressions. The operands must be instance of objects **CLASS**, **STRUCTURE** and **ALGQUERY**. The second is using the query instance of the last product done in the first step as an input operand to a select operation with a predicate that is a translation of **P(…)** – there is a wide coverage of our algebra's possible predicates to those of ODL but it is not complete. The output is then determined by a project, mapping and nesting operations. It needs to be noted that our map operand functions take one argument at a time, which could be a tuple structure. Henceforth, we are using very short syntactic representation of our operator: for example rather than **ALGEBRA[%SV(C1,"DEEPEXTENT", EQOP(…)]** we are using **SV(C1, EQOP(…))**.

```
wv(sv(
       xv(xv(xv( ...xv(en-1, en), … ,e3) ,e2), e1),
       p(…)),
    [ x1, …, xn+p ], [f1, …, fn+p] ).
```

Following a where clause in OQL select statement is the optional sorting directive; i.e. **ORDER BY**. In our algebra this is not implemented, and it is left as a query instance manipulation feature.

In OQL aggregates are done through the addition of the **GROUP BY** … **[HAVING** … **]** construct and it is placed after a where clause and before sorting directive. Variables y_i determining the partitioning and function **FF(…)** can output the partitioning instances and aggregate function on distinct attributes values in **PARTITION**; **PARTITION** is a set of attributes. The optional **HAVING** clause sifts partitions according to predicate **H(…)** satisfaction; it is expected that the **H(…)** predicate has an aggregate comparison; e.g. consider only partitions that have more than five instances (**COUNT(*)>5**). In OQL the general structure of an aggregate retrieval query follows:

```
select [distinct] ff(y1, y2,..., yn-1, yn, partition)
  from x1 in e1(x1)
       x2 in e2(x1, x2)
       ...
       xm in em(x1, x2,..., xm-1, xm)
[where p(x1, x2,..., xm-1, xm)]
[group by y1:g1(x1, ..., xm),     y2:g2(x1, ..., xm), ...,
          yn-1:gn-1(x1, ..., xm), yn:gn(x1, ..., xm)
[having h(y1, y2,..., yn-1, yn, partition)]
]
```

In our algebra the **GROUP BY** is implemented by using aggregate operator (i.e. **GV**) and possibly mapping (i.e. **WV**), and is very close to ODMG's specification. However the optional **HAVING** clause is not implemented as already stated; the aggregate filtering needs the select operator parser and evaluator to be developed further to take care of aggregate comparisons.

The following is a canonical translation for OQL's **GROUP BY** queries; properties g_1 to g_k are the grouping attributes and are a subset from x_1 to x_{n+p}, and $aggf_i$ are aggregate functions (i.e. sum, min, max, count, and avg).

```
gv(sv(
       xv(xv(xv( ...xv(en-1, en), … ,e3) ,e2), e1),
       p(…)),
   [ g1, …, gk ], [ aggf1, …, aggfk ] ).
```

13.1.10 OQL and Algebra

A common use of database algebras, at least didactically, is of a target language for declarative language queries. The ODMG's OQL is an object-oriented and declarative query language whose representation is based on "select from where" structure.

In this section we showed qualitatively two things: firstly how and how much our query and data model compare and cover with ODMG's OQL; and secondly a canonical conversion from OQL constructs into algebraic expressions composed from our operators. Therefore our algebraic expressions are closed, and at least Codd complete.

In many cases the translation is clean and effective. In other cases our handing of a concept or technique subsumes the one of OQL; for example in path expressions. Clearly the algebra developed here has an adequate and useful contribution for the procedural study of OQL queries.

The list of short falls and non-coverage is wide; this is expected as OQL standard has had a long evolution and is feature rich. In our defence even production systems do not implement OQL standard; for example the Spartan EyeDB as shortcoming in its excellent implementation. The most significant missing features in our algebra are: no multiset data type constructor; does not create objects during query processing other than related to object **`ALGQUERY`** and its instances; and not implementing the **`HAVING`** filtering clause for the aggregate operator. We repeat these are more because of time constraint than inability.

Also some issues need better treatment and these include: better treatment of the perfidious null; rethink of the late binding for deep extent queries; once multiset and array constructors are available the ODMG typing rules (and their adjustment as in Bierman and Trigoni [BIERM00], and Alagic [ALAGI00]) are revisited; finally the select operator parser should be one to one with OQL's grammar.

13.2 Query Processing and Optimisation in our Framework

In the relational data model SQL construct formed through the "select from where" are declarative queries over a relational model. These queries are based on the select, project and join operations (viz. SPJ) of relational algebra. Declarative queries have to be translated into a procedural program, or an *access plan*, that computes and then executes a plan (see figure 13.1). This translation must be correct and complete, and the translation is also expected to generate a good access plan for execution. The effort put into the translation process has to be marginal.

During the translation and execution of access plans, called query processing, there are a number of aspects that make up its context. One aspect is the *hardware available* in terms of CPU, RAM and disks arrays. Hard disks have two relevant constraints: access is paged and is at a very slow speed (approximately 1500 times slower than direct access in RAM), and a disk provides a limited transfer throughput. Another aspect is the *logical and physical design*. For example a single item access based on a key is strongly facilitated if it is indexed; otherwise an expensive serial scan is required to find the item.

In reality the expectation of the best access plan generation for any declarative SPJ query is impossible to satisfy all the time. Finding a general solution to SPJ query optimisation is not only complicated by contextual aspects but has been shown to be *NP-complete* by Ibaraki and Kameda [IBARK84]. Yet query processing and optimisation is still done with the main justification being the difference between a bad plan and a good one (even if not an optimal one) could be dramatic in terms of time and resources [IOANN96]. It is important to set and have a single optimisation goal as an objective, such as either query running time or reducing hard disk reads and writes.

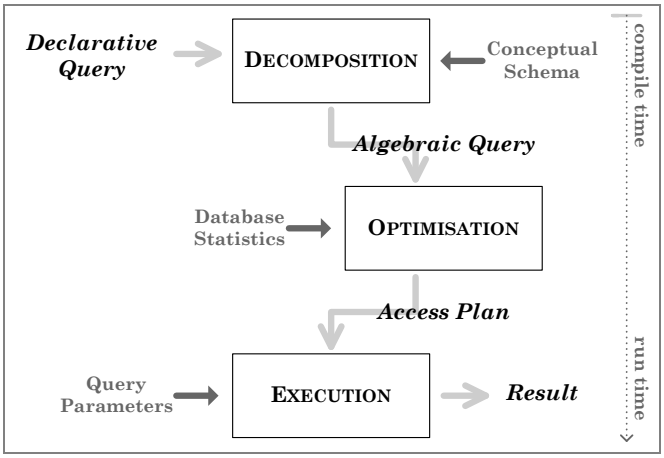


Figure 13.1: Query processing & optimisation schematic diagram

How is optimisation of declarative queries attempted? The seminal paper is Selinger's [SELIN78] on the System R optimiser. In this she introduced the technique of creating a search space of query plans, drastically pruning it for the more promising ones by applying basic heuristics, and then after estimating each of these remaining query plan's costs, choosing one which has the least of cost. The space pruning heuristics include: look only at left-deep plans for joins (for N way join reduce from enumeration from $N!$ to $N.2^{N-1}$); and choose sort-merge join over nested loop as the result maintains an "interesting order" (i.e. eliminates the sorting of a consequent sort-merge join). An intermediate step in System R optimisation prior to costing is also heuristically driven and it moves selections down and projections up a join processing tree. It has to be repeated that the chosen plan is not guaranteed to be an optimal plan as it could have been excluded from search space – neither is it possible to claim that no false negatives are pruned out.

For costing plans, the System R optimiser maintains discontinuous statistics on the state of the database; for example, the number of tuples per relation, and cardinality and selectivity of attribute values feed its costing formula. To attenuate optimisation costs, System R opts to

compile a query and save once and then run it many times until it remains valid. There are now other statistics gathering regimes, use of histograms rather than counts, more frequent collection, and sample rather than total coverage. It is worth mentioning that an early paper by Kumar [KUMAR87] says "an optimal plan for most queries is very insensitive to selectivity inaccuracies".

The System R compiler came to be called *Cost Based Optimiser* (CBO) and it introduced an important new interdependence between logical and physical perspective of database activities. For teaching purposes it makes sense to translate into relational algebra operands as query plan primitives but in actual implementation of System R, basic file and index routines are used instead. CBOs are the dominant RDBMS compiler and can be found in MS SQL Server, Oracle and Postgres SQL.

13.2.1 Query Processing and Optimisation in Deductive Databases

In deductive systems the predominant technique for query optimisation is based on 'magic sets' rewriting [BANCI86]. In Flora-2 it is implemented with tabling which the authors claim is better suited for optimisation on Prolog engines [YANGG08]. Another 'optimisation' is using main memory space to hold the object base; with the resurgence of main-memory databases and the availability of impressive amount of SSD devices it is no longer considered far-fetched, on the contrary it is becoming common.

13.2.2 Physical Query Processing and Optimisation

Today a wide array of hardware devices is available for a DBMS to manage and optimise. These include the traditional CPUs, RAM, hard disk arrays and tape. Other devices that are gaining prominence are the solid state disk (SSD), and multi-core CPUs. For example SSD have comparable read and write performance to disks but are 100 times faster in seeking [AGRAW08] and [GRAEF09]. Given that a conceptual design of a database has a huge number of possible physical implementations it is inevitable that some optimisation is required to utilise the resources available.

Currently the database designer has to add physical indicators through high level definition commands – for example the class instance person is to build a B+ tree index [KIMWO89] and [COMER79] on 'family name' attribute and a hash-based index on his 'address of residence'. Once these decisions are made and implemented then a number of consequences are fixed and leave the

query optimiser with fewer options even though an index is advantageous in many cases. Also any index created carries a cost to be kept up to date.

During an access plan design by the RDBMS the operations constructs that are seemingly not our algebraic operands; actually these are physically supported operators (or example B+ tree and Hash index). For example the algebraic select operand has some thirty different physical selects operands in Oracle. These are typically of iterative nature and examples include 'get next record', 'restart' [GRAEF09]. Also it is as important to define 'physical' functions available but which do not have any algebraic equivalent; these include disk-based sorting and temporary result storage and retrieval.

13.2.3 OODB Query Processing (QP) and Optimisation (QO)

There are issues in which object-oriented query processing differs from relational query processing. These arise from different data types available, complex object structures, class hierarchy, and methods invocation during query processing.

Given an object database schema is richer than a relational schema, then there is scope for more semantic optimisation (i.e. rewriting a query into a form that is easier to compute but returns the same answer [HAMMA80]). Use of semantic optimisation can also aid better access plan generation.

Nonetheless there are common approaches. For example both algebraic transformation (as seen earlier with our object algebra) and CBO are both a good basis for building of a query processor and optimiser for an object-oriented database. The first optimisers originate in Orion [KIMWO90C] and O_2 [ATKIN92]. Orion's approach was to reduce a query into a logical graph and its sibling leaf vertices are identified and processed (e.g. reduced) and then replaced, leading to a unique evaluation of the graph. Two heuristics were used: one for preferring indexes and the second, used if no index were found, to opt for any object clustering on disk.

Indexing on objects and convenient object placement are useful techniques available to a query processor to attempt efficient execution queries. In OODB indexing and object placement (the fixed type is called *clustering* and temporary type is called *interesting order* in System R) have more varied application as we need to index on the instance-of and *ISA* relationship (e.g. class hierarchy index) and the object composition (e.g. path expressions). Also logical identifiers are

unidirectional. What to index and how much each index costs are extremely important decisions and the two main references for costing these structures in OODB are Bertino [BERTI00] and Galdarin [GARDA95].

Class hierarchy indexes are usually implemented with B+ tree – these are robust and widely known artefacts that allow direct access and consequent sequential scans on key values (e.g. range queries). An even faster direct mechanism for indexing an object (but without consequent sequential scans – range queries) does exist and it is based on external hashing. In the case of indexes for path expressions a number of structures have been advocated for speeding up their evaluation – mainly by Bertino [BERT94] and are called nested, path and multi index.

13.2.4 Framework Support for QP and QO

There are a number of features in our framework that supplement the query processing and optimisation of algebraic queries. Clearly the algebraic operands, their supporting procedures and the object base are fundamental. While executing the queries we have utilised the data type signature of the underlying ranges found in the schema encoding; being **`CLASS`**, **`STRUCTURE`** or **`ALGQUERY`** instances. Part of the algebraic operator's execution included adding properties to the query instance that is derivable from the ranges' details; for example the query instance data-type signatures.

There are other features that our framework can provide to enable better query processing and execution. Firstly there are a number of properties attached to ranges, for example a class-instance primary-key constraint set, which can be moved to the query instance. The idea of moving the integrity constraints is not for their enforcement at query instance level but to give the query optimiser an insight into the query instance extent data properties. For example if a range of the select query has a primary key set defined on its range then there is no need to run the duplicate detection and removal effort. Not all constraints can be moved to a query instance.

A second support is to introduce features into our framework that are useful for query processing. For example we have mentioned earlier that CBO relay on database statistics to prune the sample search space. This is really required when we have a long sequence of product operations and also long-winded select predicates. The database statistics can either be part of a range

definition, for example part of a class instance definition, or part of an in-built statistical data dictionary.

13.2.5 QP and QO – Reduction of Products

One of the most often cited advantages of expected query optimisation in object-oriented queries is the reduction in costs of joins. For example consider the following query construct in OQL which returns a set of student names taught by a full professor:

```
select distinct x.sname
  from students as x,
       x.takes as y,
       y.taught_by as z
 where z.rank = "full professor";
```

A canonical translation of this into our algebra follows (note the abbreviated naming of our operands) and it is characterised with two products, three selects and a project. Clearly this is not what is anticipated.

```
pv(
   sv (
        sv(
            xv (
                 sv (
                      xv ( student, section ),
                      epop( ipe,takes,loid,section,loid) ),
                 professor ),
            epop( ipe,taught_by,loid,professor,loid)  ),
         eqop( ipe,rank,string,"full professor",string )
       ),
     [sname] ).
```

How can we reduce the number of products and selects? An effective technique is to encode parts of the OQL query into what is called a *connection graph* where each "from" expression and "where" conjugant are depicted into a graph as a node. The graph's edges represent the bindings between the nodes or to literals (e.g. in the where clause). During the building of the connection graph additional data from the underlying framework is extracted and shown as shaded labels (e.g. implicit class names and a property's cardinality constraint). The following build-up represents the connection graph for the above OQL.

It is relatively easy to identify any leading node (have no incoming edge) and in this case it is **STUDENT**. If all upper limits of the connections are '1' except the last then follow the path to an end node (e.g. **STRING="…"**) to generate the following path expression: **X.TAKES.TAUGHT_BY.RANK = STRING("…")**. Now we can re-write the original OQL as:

```
select distinct x.sname
  from students as x,
 where takes.taught_by.rank = "full professor";
```

This has now been reduced to a select, albeit with logical identifier chasing, and a project.

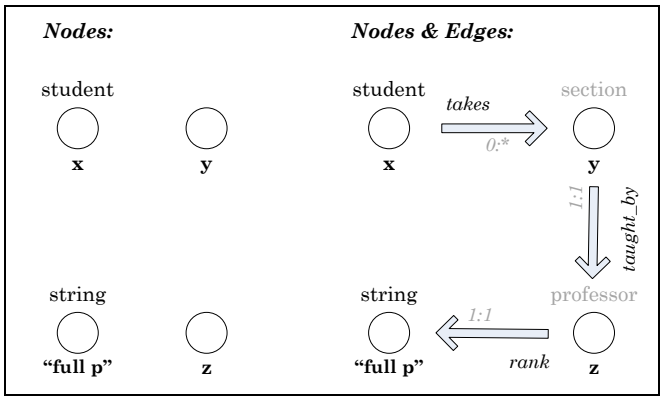


Figure 13.2 - Building a connection graph for an OQL query

If in a path there is an internal node with an upper limit of more than '1', as is actually the case in figure 13.2, then the path needs to be broken into two. (This is a limitation of OCL's use of paths – see section 13.1.3). The OQL select statement follows:

```
select distinct x.sname
  from students as x,
       x.takes  as z
 where z.taught_by.rank = "full professor";
```

The connection graph can tell us a number of things: for example, if the graph is made of two networks then there really is a Cartesian product; if there is a directed loop in a path expression then recursion is possible. Since our algebra does not support recursion then there are issues to resolve prior to executing the query.

Another level of path expression handling from a connection graph is a consideration of physical indexes present. For example if an index exists for **STUDENT.TAKES.TAUGHT_BY** then during query optimisation the path expression is broken into a conjunction of two: **X.TAKES.TAUGHT_BY AS Y AND Y.RANK**.

13.2.6 QP and QO – View Materialisation

Materialised views are data views and query results that are stored [VALDU87]. Furthermore current DBMSs have facilities to keep the materialised view up-to-date or completely refresh on demand. Dependence can exist between materialised views as a view can be defined in terms of other views. In which case the query processing does not need to re-compute the dependent views as these are already stored; consequently query processing time and disk bandwidth is drastically reduced. Query optimisation by materialised views is popular in OLAP and data warehousing queries. The main issue of course is what, when, where and how to materialise views [HARIN96].

In our paper, Vella and Musu [VELLA00], we presented a break-even analysis between materialising and re-query of a sample of data warehouse queries.

In our framework query results are materialised. It is natural then to think about using this aggressive optimisation technique when optimising our algebraic queries.

A popular OLAP query is rollup; i.e. applying consecutive summaries onto a data cube. The dimensions of a data cube are the partitioning attributes of aggregate operator. The content of each cell in the cube is the result of the aggregating function. Consider the following rather simple cube whose dimensions are customer, product, and region. The content of each cell is the sum of all sales relative to **CUSTOMER**, **PRODUCT**, and **REGION** recorded in the previous financial year. Assume we had a large number of transactions and typically the cube is sparse (i.e. not all combinations exists). The cube is built by the following aggregate query:

```
gv(sales,"deepextent",[customer,product,region],[amount],[sum]).
-- remark the algquery instance is id. by gv(sales,"deepextent",8811)
```

Assume a rollup is requested and the following is submitted:

```
gv(sales,"deepextent",[product,region],[amount],[sum]).
```

The query instance does compute the proper answer but all the source data, i.e. in **SALES**, is re-visited. This is not necessary at all because the following query, whose range is changed as a result of the first query, computes the same answer:

```
gv(
  gv(sales,"deepextent",8811),
  "deepextent",
  [product,region],
  [sumamount],
  [sum]).
```

To realise this optimisation technique the aggregate operator needs to do the following: first find an instance of **ALGQUERY** whose identifier functor unifies with **GV(SALES,"DEEPEXTENT",?_)**; second and if so is the current partitioning list of properties a subset of the **GV(…).GROUPBY** list of partitioning list of properties; third if so do the attribute and aggregating function match (i.e. examine properties **AGGOUT** and **FUNCAGGOUT**). If positive then enough is at hand to rewrite second query above into the third.

Therefore on the first query, we can use the result to compute seven other partitions; i.e. on {a,b,c} we can materialise on partitions {a,b}, {a,c}, {b,c}, {a}, {b}, {c}, { }. If view materialisation is

structured it is best to stop at first level (i.e. two element sets), and compute the second (i.e. one element sets) from the first. The same procedure applies for the third level.

In our previous paper [VELLA00] it was found that widening aggregate function list is generally useful and gives the materialised view higher reusability. In our previous example, if other than aggregate function sum we compute in parallel count, max, and min, then the materialised view would work out these and the average too (as average is derived from sum divided by count).

There are other view materialisation techniques and all are relatively easy to implement. One design problem is to ensure that queries are monotonic increasing; in which case view materialisation might not be possible or best indicated technique.

13.2.7 QP and QO – Simplification of the Select Predicate

In the presentation of the select operator we commented on the structure of its predicate. We also delayed the manipulation of the predicate even when we stated its properties; e.g. commutativity of conjugants. The justification being that the number of equivalent rewrites possible quickly expands. In fact the basic technique is to write predicate in a standard form, actually conjunctive normal form (CNF) is preferred, and then the predicate tree is traversed and certain patterns, if present, are developed. That is these patterns become unmoveable while the others can be juxtaposed. The pruning of the search space could also be aided by the database statistics, e.g. conjugants with high selectivity given priority to compute first, and knowledge of physical artefacts (e.g. presence of an index that covers a conjugant).

During the traversing of the predicate tree, which by now is in CNF, one can test for patterns that do not need access to the object collection to compute. For example if property **SNAME** is a single valued string then two conjugants of the form **SNAME='JACK'** and **SNAME='RICK'** are false

Another optimisation method which can be applied at this point falls under the semantic optimisation title. In our framework we have constraints attached to **CLASSES** and **STRUCTURE** instances. Two examples are the not null and the check constraints. In the case of a select both of these constraint instances related to the select range remain invariant and could be copied to the query instance, much as we copy the data-type signature.

Say **SNAME** has a not null constraint, then any part of the predicate of the form **SNAME IS NOT NULL** is always true. On the contrary a conjugant of the form **SNAME IS NULL** is always false. In either case the predicate can be simplified before evaluating any data.

Similarly say we have a check constraint on **SNAME** that insists that the string has to be upper case, and then any string comparison with **SNAME** that is not in caps is false. Again there is no need to look at any of the target collection.

The following table, i.e. table 13.5, describes what and how constraints can be copied from range collections to query instances for each algebraic operator. In some cases the constraint can be asserted even when not present in the algebraic operator operand; e.g. the not null constraint in the aggregate operator.

Algebraic Operator	**Not null constraint (pre / action / post condition)**	**Check constraint (row level) (pre / action / post condition)**
union	if in both classes / ok / none	if in both classes / ok / none
diff	if in both classes / ok / none	if in both classes / ok / none
product	none / ok / none	none / ok / none
project	if in project list / ok / none	if in project list / ok / none
select	none / ok /none	none / ok / none
aggregate (on grouping) **aggregate (result)**	none / ok / if not present assert for each attr none / none / assert	none / ok / none none / none / none
Map (1 to 1)	none / ok /none	none / none / none
nest (on grouping) **nest (result)**	none / none / assert for each attr none / none / assert for each attr	none / ok / none none / none / none
unnest (on repeating)	none / ok / if not present assert for each attr	none / ok / none
unnest (result)	none / none /assert for each attribute	none / none / none

Table 13.5: Moving not null and check constraints to **ALGQUERY** instances

13.2.8 QP and QO – Duplicate Elimination

During the description of the algebraic operators the duplicate detection and removal procedure was required in many operators; like in project and select operators. Duplicate detection and removal is a computationally expensive operation. This does not need to be always the case. Consider a select operation over a class instance **STUDENT** which has a primary key set on

property **STUDENT_ID**. The select predicate does not change the value of any property and therefore any result of the select operation still respect the primary key constraint. Consequently there is no need to run a duplicate detection and removal on the select result set if primary key set is present.

To introduce this optimisation into our framework there is a need the following additions: first the copying of integrity constraint, according to the table 13.6 below for each algebraic operator, from the target class instance into the relative query instance. This could be done with reification rules instantiation. Second is check whether the duplicate procedures need calling; this could be done in the operator logic or in the duplicate procedures themselves.

Algebraic Operator	**Primary key & Cand. Key constraint (pre / action / post condition)**
union	none / none / none
diff	none / none / none
product	if exists in both targets / none / concat both pks to assert new pk
project	if in project list / ok / none
select	none / ok / none
aggregate (on grouping) **aggregate (result)**	none / none / assert pk of all attr none / none / none
Map (1 to 1)	if map is 1:1 / ok / none
nest (on grouping) **nest (result)**	none / none / assert pk of all attr none / none / none
unnest (on repeating)	none / none / assert pk of all attr
unnest (result)	none / none / none

Table 13.6: Moving primary key and candidate key constraints to **ALGQUERY** instances

In table 13.6 the pre-action, action, and post-action for copying the primary key set and candidate key are given. It is important to note that union and difference operators remain impervious to optimisation. It can be remarked though, that if the union compatibility is based on self and sub-

type and a primary key set is present in the parent class then duplication elimination avoidance is, in some cases, possible.

More elaborate semantic optimisation is possible in the presence of a functional dependency. From section 7.4.5 in integrity constraint chapter an example shows how query predicates demanding a product and a select operation is reduced, in the presence proper functional dependency, into a select operation only.

13.2.9 Query Processing and Optimisation

Query optimisation is still following the seminal work of Selinger *et al.* [SELIN78] from the early eighties with CBO and search space pruning its distinctive features. We have indicated works and studies that extend Selinger's work into object-oriented databases too. We have shown how our algebraic expressions can built around this model too. Keeping details of a query in a query instance make some sophisticated optimisation techniques easier and less likely to miss; for example in materialising aggregates keeping the partitioning and aggregate function details makes the technique easy to identify and use. Furthermore we have shown how our framework not only integrates with best practices in query optimisation but is also capable to implement specialised optimisations techniques: these include view materialisation, join reduction, and semantic optimisation. Another important aspect was the recognition of physical artefacts, for example index systems, by the framework through which optimisation can be enhanced and improved.

During this section we also gave details on how to move constraints from schema features into query instances. The tables, i.e. 13.5 and 13.6, are detailed and offer an interesting array of possibilities for optimisation.

Implementations of these routines are all in Flora-2 and form part of our framework. Furthermore we assert that the computational cost of our query optimisations is indeed marginal as it only involves the schema rather than the actual object ranges of the query expression.

13.3 Summary

An important declarative query language for object databases is ODMG OQL. The last published version, i.e. 3, is based on a long history and has many features. In this chapter we gave a qualitative study on how OQL declarative constructs can be converted into an expression

composed of our algebraic operators through a procedure we outlined. Although the procedure is naïve some translation tricks are introduced to make it more effective. Also a comparison of ODMG's query model and ours was undertaken; the results are positive. Nonetheless a number of ODMG features are missing; these are mostly time related rather than difficulty to implement into our query model. The most notable missing features are data type constructors, e.g. multi-set, and not implementing OQL having clause in group by.

Once a mapping from a declarative OQL to our procedural constructs is made then our algebraic expressions can be manipulated to find an expectedly lower cost access method. This query optimisation is essential and it is expected that traditional methods are implementable in our framework. In this section we also showed examples of aggressive and specialised techniques in query optimisation such as semantic optimisation and view materialisation. Also the use of data model constructs like integrity constraints, to address computationally expensive operations was explained. In fact this is possible when a query instance inherits not only the data type signature but also the constraints found in its operands.

This chapter has shown the importance of integrating more tightly the translation from conceptual schema design right up to query processing and optimisation. For example data type signatures inferred in the schema mapping are used in query optimisation. Another example is the primary-key declaration at the EERM level is still in use during query optimisation. The push for object homogeneity helps organise development and dependences between a systems components even if these are at a schema level. Also the technique of leaving details at a meta level of operations that converted or created an object is effective and makes application development easier to manage. An example is query optimisation with view materialisation uses the details of the original query to compute other but related queries with a lighter footprint.

Chapter 14

Conclusions

14 – Conclusions

The efficacy of our goal to tightly couple conceptual design, object-oriented data modelling and object-oriented query modelling is best seen in Chapter thirteen, i.e. in query processing and optimisation. Firstly, a declarative query is translated into an algebraic equivalent. Secondly, although the generic optimisation examples are in join reduction, view materialisation and semantic rewriting, in each case there is an important input from the data modelling constructs during the query optimisation phase. Thirdly, our decision to materialise a query into an object and its result set an instance of the query object created allows for the query's instances to retain any applicable data modelling constructs from the target set, e.g. primary key constraint. Therefore the algebra does not only have closure but if the result set is in turn involved in another query then the optimisation can still consider any of the 'inherited' constraints.

The query model's effective dependence on the data model is a result of the careful and viable design decisions taken when surveying database technology and object-oriented themes and variations. Furthermore the query model has strong object-oriented features too: e.g. a query being an object and having a class role too, a query has a data type signature for its extent, and a query operates on either the deep or shallow extent of its target set. Finally the query model has both declarative and procedural languages.

The persuasive framework developed with Flora-2 integrated conceptual design, data and query modelling, and data typing and inference. The framework is not only indispensible to translate from one 'language' into another, but also very good in aiding development. For example, when translating from conceptual to logical a framework's method determines how best to convert a conceptual n-ary relationship into its proper set of constraints in the logical model. Finally the framework was used to make good for any shortages in other structures or tools. For example Flora-2 allows data type specifications in a program but these are not typed check on a program's evaluation. Consequently data type checking and inference has to be coded and invoked in the framework.

Development of the framework with Flora-2 has another important facet: each method developed, e.g. the algebraic operators, is coded in declarative constructs. This effectively makes the semantic description stronger.

14.1 Strengths and Weaknesses

The strength of this work, in summary, lies with the use of the object-oriented paradigm to construct a tightly-coupled data and query model for an object-oriented database. The query model is interpreted in either a declarative or a procedural query; the latter is expressed in our specially constructed algebra. Our framework manages, controls, and is able to improve aspects in the development of an object-oriented database. Furthermore we see our framework as a basis to other DBMS related facilities; for example, cost based query optimisation.

There are a number of weaknesses that need addressing and the following points elucidate a few:

Firstly, a query's result set is currently of a homogenous data type even if the query target set is the deep extent of a class. For taking up of heterogeneous result set, all of our algebraic operators need to be revised and it is expected that extensive knowledge in data type checking and inference at run-time need to be implemented. Most of these techniques have been developed and verified in the very recent years; i.e. post 2005.

A related second weakness deals with the limited range of available data type constructers, i.e. tuple, set and list, in our data and query model. The techniques used here, for example F-bounded polymorphism in type checking list and *ISA*, are well known and have been developed and proven effective by the year 2000. An example of a missing data type constructor is a bag (i.e. multi-set). Again this entails a major rewrite of all algebraic operators to handle the different semantics of each data type constructor.

The third weakness is found in the query optimiser; as no cost-based model is implemented it is therefore impossible to give quantitative cost estimates on of query plans.

The fourth weakness is the total absence of a transaction model.

A fifth weakness, related to the EERM, is that its more advanced constructs require extensive support by transitional constraints. The research community, in conceptual design and DBMS facilities, should address this too.

A sixth weakness is the tenable perception, or specifically the effective benefit, of materialising the query result as pointless because of its currency and thee computationally heavy copying of result sets. As much as we accept this criticism, we do repeat our argument that for some domains of discourses this technique is useful.

14.2 How have the tools fared

The simplest tool used was GraphVIZ. The package is flexible and neat; it also has a gentle learning curve. There was no requirement of ours that GraphVIZ did not cater for.

EyeDB, an OODBMS, was a pleasant surprise and it does work. Even though its implementation is not 100% compliant with the ODMG standard it still offers a good spectrum of ODMG and OODB facilities. It does not have any auxiliary tools one expects.

Flora-2 is a very good system. User interface, other than its CLI, is in Emacs. We feel that this project, if proof was ever needed, shows that F-logic has an interesting and effective set of features and that Flora-2 implementation offers designers and developers with facilities that enable large-scale projects to be realised. An extensive list of features have been used to implement our framework: identity equality, reification of rules and their evaluation, parsing a string and evaluating it on the knowledge base, path expressions, complex identifier structure, and updating the knowledge base. Nonetheless it does have a learning curve.

14.3 Future Additions

There are many points that with hindsight could have been done better. The following is a list of what the next efforts should be or what has to be addressed.

- EERM drawing has to resolve some diagrammatic issues (e.g. aggregation);

- EERM encoding in the framework should add some heuristic checks, for example to attempt flagging connection traps;
- For EERM to ODL mapping to achieve higher completeness we need to encode transitional actions and constraints associated with sophisticated design requirements (e.g. aggregation) in the framework, and have facilities in ODL (e.g. triggers and view update mechanisms);
- The framework needs to have sophisticated methods to check consistency and redundancy within the integrity constraints collection;
- The framework needs a rewrite of procedures involving path expressions;
- Better and cleaner use of 'self' and 'null' in our framework;
- EyeDB – addition of data dictionary views (otherwise automated conversion of OQL queries to algebra are hampered);
- EyeDB – addition of the equivalent of SQL's "ALTER TABLE" commands;
- Data type check the schema objects too;
- The algebra should handle and output heterogeneous sets and then bags of. Consequently use the better of data typing rules which have surfaced in the last ten years;
- The algebra needs to align closer to more OQL features (e.g. select predicate and OQL where clause);
- QP-QO needs abstraction of physical database operations and artefacts (e.g. indexes for path expressions);
- QP-QO in the framework needs to build object database statistics profile and consequently are used as a basis in cost based optimization;
- QP-QO simpler techniques that are known to be popular and cost effective are introduced.
- The framework could be adopted to 'export' to other data modelling languages, including graph and document based models found in some NoSQL systems.

Appendix –Bibliography

ABADI91 Abadi, M.; Cardelli, L.; Pierce, B.; and Plotkin, G.; "Dynamic Typing in a Statically Typed Language", ACM Transactions on Programming Languages and Systems, V(13)2, 1991, pp.237-268

ABITE89 Abiteboul, S.; Kanellakis, P., C "Object Identity as a Query Language Primitive", Proceed. of the 1989 ACM SIGMOD, Oregon, SIGMOD REC, Vol. 18, No. 2; 1989; p.p.159-173

ABITE89B Abiteboul, S.; Kanellakis, P., C.; "Object Identity as a Query Language Primitive", ACM SIGMOD1989, Oregon, SIGMOD REC, Vol. 18, No. 2, 1989, Ed. Clifford, J.; Lindsay, B.; Mair, D.; pp.159-173

ABITE93A Abiteboul, S.; Beeri, C.; "On the Power of Languages for the Manipulation of Complex Objects", I.N.R.I.A., Cedex France and Department of Computer Science, The Hebrew University, Israel, Technical Report Feb., 1993 Second Edition, p1-76

ABITE95 Abiteboul, S.; Hull, R.; Vianu, V.; "Foundations of Databases", Addison-Wesley, 1995, pp. 685

ABITE95B Abiteboul, S.; den Bussche, J., V.; "Deep Equality Revisited", Proceed. of Conf. on Deductive and Object-Oriented Databases - DOODs,(3rd), 1995, Singapore, 1995, pp.213-228

AGRAW08 Agrawal, R.; et al, (2008), "The Claremont report on database research", ACM SIGMOD Record, Volume 37, Issue 3 (September 2008) Pages 9-19

AHOAL79 Aho, A., V.; Ullman, J., D.; "Universality of Data Retrieval Languages", Conf. Rec. of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, 1979, pp. 110-120

AHOAV79 Aho, A, V; Beeri, C; Ullman, J, D; The theory of joins in relational databases, ACM Tran. On Database Systems, v4(3), 1979, pp. 297-314

ALAGI99 Alagic, S.; "Type-Checking OQL Queries In the ODMG Type Systems", (2000), ACM Transactions on Database Systems, Vol. 24, No. 3, September 1999, Pages 319-360

AMERI87 America, P.; "Inheritance and Sub-typing in a Parallel Object-Oriented Language", ECOOP'87, France, LNCS 276, Springer-Verlag, 1987, pp.234-242

AMERI90B America, P; "Designing an object-oriented programming language with behavioural subtyping", Foundation of Object-Oriented Languages, Rex School/Workshop, The Netherlands,1990, SV-LNCS#489, pp.60-90

ANDRE90 Andrews, T; Harris, C; Duhl, J; The Ontos Object Database, TR, Ontologic Inc, 1990

ANSIT86 ANSI, "SQL Standard - INCITS/ISO/IEC 9075-86 ", ANSI, 1986

AQUIL97 Aquilino,D.; Asirelli, P.; Renso, C.; Turini, F.; "Applying Restriction Constraints to Deductive Databases", Ann. Math. Artif. Intell. V19(1-2):, 1997, pp.3-25

ATKIN87 Atkinson, M; Buneman, P; Peter, O; "Types and Persistence in Database Programming Languages", ACM Computing Surveys, V(19)2, 1987, pp.105-190

ATKIN90 Atkinson, M.; DeWitt, D.; Maier, D.; Bancilhom, F.; Dittrich, K.; and Zdonik, S.; "The Object-Oriented Database System Manifesto", Proceed. of Conf. on Deductive and Object-Oriented Databases - DOODs,(1st), 1989, pp.223-240

ATKIN92 Atkinson, M., P.; Bancilhon, F.; DeWitt, D., J.; Dittrich, K., R.; Maier, D.; Zdonik, S., D.; "The Object-Oriented Database System Manifesto", 'Building an Object-Oriented Database System, The Story of O2', 1992, pp. 3-20

ATKIN95 Atkinson, M.; Morrison, R.; "Orthogonally persistent object systems", The VLDB Journal, V(4) # 3, 1995, pp. 319-402

BAKER97 Baker, Sean; CORDA: Distributed Objets Using Orbix, ACM Press, Addison-Wesley, 1997

BANCI85 Bancilhon, F.; "A note on performance of rule based systems", Technical Report MCC, 1998

BANCI86 Bancilhon, F.; Maier, D.; Sagiv, Y.; Ullman, J., D.; "Magic sets and other strange ways to implement logic programs", ACM PODs, 1986, pp. 1-15

BARBI03 Barbier, F.; Henderson-Sellers, B.; "A Survey of UML's Aggregation and Composition Relationships", L'OBJET,V5(3/4); 1999

BATIN92 Batini, C.; Ceri, S.; Navathe, S.; "Conceptual database Design", Benjamin Cammings; 1992, pp. 470

BEERI90A Beeri, C.; "A Formal Approach to Object-Oriented Databases", Data & Knowledge Engineering, 5 (1990), North-Holland, pp.353-382

BEERI99 Beeri, C.; Thalheim, B.; "Identification as a primitive of database models",in "Fundamentals of information systems" Ed. Polle, T.; Ripke, T.; Schewe, K.-D., Kluwer, 1999

BENED08 Benedikt, M.; Koch, C.; "XPath Leashed", ACM Computing Surveys, V 41(1), 2008

BEREN95 Berenson, H.; Bernstein, P.; Gray, J.; Melton, J.; O'Neil, E.; O'Neil, P.; "A Critique of ANSI SQL Isolation Levels ";SIGMOD, 1995

BERLI90 Berlin, L.; "When Objects Collide: Experiences with Reusing Multiple Class Hierarchies", ACM - ECOOP/OOPSLA'90 Proceed., pp.181-193

BERTI00 Bertino, E.; Catania B.; Filippone, A.; "An index allocation tool for object-oriented database systems", Software Practice and Experience, V30, 2000; p.p. 973-1003

BERTI89 Bertino, E.; Won Kim; "Indexing Techniques for Queries on Nested Objects", IEEE Trans on Knowledge and Data Engineering, V(1)2, 1989, pp.196-214

BERTI91 Bertino, E., Martino, L.; "Object-Oriented Database Management Systems: Concepts and Issues", IEEE Computer, April, 1991, pp.33-47

BERTI94 Bertino E.; "Index configuration in object-oriented databases", VLDB Journal, 1994, V3(3), 1994; p.p. 355-399.

BIERM00 Bierman, G.; Trigoni, N.; (2000), " Towards a Formal Type System for ODMG OQL", University of Cambridge, Computer Laboratory, TR 497

BOBRO86 Bobrow, D., G.; Kahn, K.; Kiczales, G.; Masinter, L.; Stefik, M.; Zdybel, F.; "Common Loops: Merging Lisp with object-oriented Programming", OOPSLA'86 Conf. Proceed., ACM SIGPLAN, N21.11, 1986, pp.17-29

BONNE94 Bonner, A., J.; Kifer, M.; "An overview of transaction logic. Theoretical Computer Science", V133, 1994, p.p. 205-265

BOOCH00 Booch, G.; Jacobson, I.; Rumbaugh, J.; "OMG Unified Modeling Language Specification"; OMG consortium; Version 1.4; 2000

BOOCH99 Booch, G.; Rumbaugh, J.; Jacobson, I.; "Unified Modeling Language User Guide", Addison-Wesley, 1999

BUNEM96 Buneman, P.; Davidson, S., B.; Hillebrand, G., G.; Suciu, D.; “A Query Language and Optimization Techniques for Unstructured Data”; SIGMOD Conference 1996, pp. 505-516

BRACH83 Brachman, R.; "What is-a is and isn't?", IEEE Computer, V16(10), 1983, pp.80-93

CANNA93 Cannan, S., J.; Otten, G., A., M.; "SQL - The Standard Handbook; based on the new SQL standard (ISO 9075:1992(E))"; McGraw-Hill Book Co, 1993

CANNI89 Canning, P.; Cook, W.; Hill, W.; Olthoff, W.; Mithcell, J., C.; "F-Bounded Polymorphism for Object-Oriented Programming", Proceed. of Functional Programming, Languages and Computer Architecture, 4th Conf., England, 1989, pp.273-280

CARDE84 Cardelli, L.; "A Semantics of Multiple Inheritance", Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceeding Ed. G. Kahn, D.B. MacQueen, G. Plotkin; Springer-Verlag, LNCS#173, 1984

CARDE85 Cardelli, L., Wegner, P.; "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computer Surveys, V17(4), 1985, pp.471-522

CARDE88 Cardelli, L.; "A Semantics of Multiple Inheritance", Information and Computation, vol. 76, 1, 1988, p.p.138-164

CATTE00 Cattell, R., G., G.; Barry, D., K.; Ed., (2000), The Object Database Standard: ODMG-3,0, Morgan-Kaufmann Pubs., 2000

CATTE94 Cattell, R., G., G.; Ed., (1994), The Object Database Standard: ODMG-93, Morgan-Kaufmann Pubs., 1994

CERIS89 Ceri, S.; Gottlob, G.; Tanca, L.; "What You Always Wanted to Know About Datalog (And Never Dared to Ask)", IEEE Transactions On Knowledge And Data Engineering, 1989, Vol. 1, No. 1, p.p.146-166

CHAMB81 Chamberlin, D., D.; Astrahan, M., A.; Blasgen, M., W.; Gray, J.; et al , "A History and Evaluation of System R",. Communication of the ACM 24(10), pp. 632-646, 1981

CHANG88 Chang, C.; "On the evaluation of queries containing derived relations in relational databases", In Gallaire et al. eds, Advances in Database Theory,v1,Plenum Press, 1981,pp.235-260

CHENP76 Chen, P.; "The Entity Relationship Model - Toward a Unified View of Data", Trans. on Database Systems, V(1).1, 1976, pp. 9-36

CHENW89 Chen, W.; Kifer, M.; Warren, D.,S.; "HiLog: A First-Order Semantics for Higher-Order Programming Constructs", 2nd International Workshop on Databases Programming Languages, 1989, p.p. 1090-1114

CODDE70 Codd, E., F.; "A Relational Model of Data for Large Shared Data Banks", Comms of the ACM, V13 (6), 1970, pp. 377-387

CODDE72 Codd, E., F.; "Relational Completeness of Data Base Sublanguages", In: R. Rustin (ed.): Database Systems: Prentice Hall, 1972, pp. 65-98

COLME96 Colmerauer, A.; Roussel, Ph.; "The birth of Prolog", in History of programming languages, 1996, ACM / Addison Wesley

COMER79 Comer, D.; (1979), "Ubiquitous B-Tree", ACM Computing Surveys, V 11(2), p.p.121-137

COOKW92 Cook, W., R.; "Interfaces and Specifications for the Smalltalk-80 Collection Classes", OOPSLA'92 Conf. Proceed., Canada, ACM SIGPLAN, N27(10), pp.1-15.

COOPE06 Cooper, E.; Lindley, S.; Wadler, P.; Yallop, J.; “Links: web programming without tiers”, in the proceedings of Formal Methods for Components and objects 2006, LNCS 4709, pp. 266-296.

DAHLO66 Dahl, O.-L.; Nygaard, K.; "SIMULA - An Algol-based Simulation Language", Communications of the ACM, v9, 1966, n9, pages 671-678

DANFO88 Danforth, S.; Tomlinson, C.; "Type Theories and Object-Oriented Programming", ACM Computer Surveys, V(2)1, 1988, pp.29-72

DASSK90 Das, S, K; Willians, M, H; 'Extending integrity maintenance capability in deductive databases', In Proceed. Of the UK ALP-90 Conference, England, 1990, pp.75-111.

DELOB95 Delobel, C.; Lécluse, C.; Richard, P.; "Databases - from relational to object-oriented systems"; International Thomson, 1995, pp. 382

DEUXO91 Deux, O.;"The O2 system", Cattell, R., G., G.; Ed. "Next-Generation Database Systems", Comms. of the ACM, 1991, V(34)10, pp. 30-33

DINNA95 Dinn, A.; Paton, N., W.; Williams, M., H.; Alvaro A. A.; Barja, F., M., L.; "The Implementation of a Deductive Query", in Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95, 1995

ELMAS10 Elmasri, R.; Navathe, S., B.; "Fundamentals of Database Systems", Addison-Wesley, 2010, 6th edition, pp. 1200

ELMAS85 Elmasri, R.; Weeldreyer, J., A.; Hevner, A., R.; "The Category Concept: An Extension to the Entity-Relationship Model"; Data Knowledge Engineering; V1(1); 1985, pp. 75-116

ENDER72 Enderton, H., B.; A Mathematical Introduction to Logic, Academic Press, 1972

FROHN94 Frohn, J.; Lausen, G.; Uphoff, H.; "Access to Objects by Path Expressions and Rules"; VLDB, 1994; pp. 273-284

GALLA78 Gallaire, H.; Minker, J.; (Ed.s); "Logic and Data Bases (Symposium on Logic and Data Bases)", Plenum Press, 1978

GARDA95 Gardarin, G., Gruser, J., R.; Tang, Z., H.; (1995), "A cost model for clustered object-oriented databases", Proceed. of the VLDB, p.p. 323-334.

GOLDB88 Goldberg, A.; Robson, D.; "Smalltalk-80: the language and its implementation", Addison-Wesley, 1988

GOSLI05 Gosling, J.; Joy, B.; Steele, G., L.; Bracha, G.; "The Java Language Specification", 3rd ed., Addison-Wesley, 2005

GRAEF09 Graefe, G.; (2009), "The Five-Minute Rule 20 Years Later: Revisiting Gray and Putzolu's famous rule in the age of Flash", Comms. of the ACM, V52(7), p.p. 48-59

GRAYJ92 Gray, J.; Reuter, A.; "Transaction Processing: Concepts and Techniques", Morgan Kaufmann. 1992

GUTTA80 Guttag, J.; "Notes on Type Abstraction", IEEE Tran. On Software Engineering, 1980, V6(1), pp.12-23

HAMMA80 Hammer, M., M.; Zdonik, S., B.; "Knowledge based query processing", In Proc. of VLDB 1990; pp. 137-147

HARIN96 Harinarayan, V.; Rajaraman,A.; Ullman, J., D.; "Implementing data cubes efficiently";. In: Proc. of SIGMOD, 1996; pp. 205-216,

HEIMB85 Heimbigner, D.; McLeod, D.; "A Federated Architecture for Information Management",ACM Trans. Information Systems, V3(3), 1985, pp. 253-278

HEJLS10 Hejlsbers, A.; Torgersen, M.; Wiltamuth, S.; Golde, P.; "The C# Programming Language", 4th edition, Addison-Wesley, 2010, pp. 864

HEWIT73 Hewitt, C.; Bishop, P.; Steiger, R.; "A Universal Modular ACTOR Formalism for Artificial Intelligence", IJCAI 1973, pp. 235-245

HUANG11 Huang, S., S.; Green, T., J.; Loo, B., T.; "Datalog and Emerging Applications: An Interactive Tutorial", ACM SIGMOD'11; 2011 ,

HULLR87 Hull, R.; King, R.; "Semantic Database Modelling: Survey, Applications, and Research Issues"; ACM Computing Surveys, V.19(3), 1987, pp. 201-260

IBARK84 Ibaraki, T.; Kameda, T.; "On the Optimal Nesting Order for Computing N-Relational Joins", ACM Transactions on Database Systems, V9(3), 1984, p.p. 482-502

IBMAC93 IBM, "IBM Dictioary of Computing", McGraw-Hill, 1992, pp. 758, (Available online)

INMON02 Inmon, W., J.; "Building the Data Warehouse"; Wiley, 2002, pp. 356

IOANN96 Ioannidis, Y. I.; (1996), "Query optimization ", ACM Computing Surveys, V28(1), p.p.:121-123

JAMIL92 Jamil, M., H.; Lakshmanan, L., V., S.; "ORLOG: A Logic For Semantic Object-Oriented Models", n Proc. 1st Int. Conference on Knowledge and Information Management, 1992, p.p. 584-592

KACIA94 Ait-Kaci, H.; Podelski, A.; Smolka, G.; "A Feature Constraint System For Logic Programming With Entailment", Theoretical Computer Science, 1994, Vol.122, No.1-2, p.p.263-283

KANNE92 Kanellakis, P., C.; Lécluse, C.; Richard, P.; "Introduction to the Data Model. Building an Object-Oriented Database System", in "Introduction to Object-Oriented Database Systems", Morgan Kaufmann,1992, pp. 61-76

KENTW91A Kent, W; "Important Features of IRIS OSQL", Computer Standards & Interfaces, 1991,V(13)1-3, pp.201-206

KHOSH86 Khoshafian, S., N; and Copeland, G., P.; "Object Identity", ACM - OOPSLA'86 Proceed., pp.406-415

KIFER90 Kifer, M.; Wu, J.; (1990), "A First-Order Theory of Types and Polymorphism in Logic Programming", Department of Computer Science, SUNY at Stony Brook, TR 90/23

KIFER92A Kifer, M.; Won Kim; Sagiv, Y.; "Querying Object-Oriented Databases (Extended version)", ACM SIGMOD Conf. on Management of Data, San Diego, CA, 1992, pp.393-402

KIFER95 Kifer, M.; Lausen, G.; Wu, J.; "Logical Foundations of Object-Oriented and Frame-Based Languages", Journal of the ACM, May, 1995, v.38 n.3, p.p. 619-649

KIMWO89 Kim, W.; Kim , K.-C.; Dale, A.; (1989), "Indexing techniques for object-oriented databases", in Object-oriented concepts, databases, and applications, ACM Press, New York, NY, 1989

KIMWO89G Kim W.; Ballou, N.; Chou, H.-T.; Garza, J., F.; "Features of the ORION Object-Oriented Database System"; in Won K.; Lochovsky, F., H.; Ed.s, "Object-Oriented Concepts, Databases and Applications", ACM Press, Addison-Wesley, 1989; p.251-282

KIMWO90C Won Kim; Garza, J.F.; Ballou, N.; Woelk, D.; "Architecture of the ORION next-generation database system", IEEE Transactions on Knowledge and Data Engineering, V(2)1, March 1990, pp. 109-24

KIMWO90D Won Kim; "Introduction to object-oriented databases", Massachusetts Institute of Technology, 1990

KOWAL74 Kowalski, R.; "Predicate Logic as Programming Language", in Proceedings IFIP Congress, Stockholm, p.p. 569-574, 1974

KOWAL79 Kowalski, R., A.; "Algorithm = Logic + Control", Communication of the ACM, v22(7), 1979, pp.424-435

KOWAL87 Kowalski, R; Sadri, F; Soper, P; "Integrity Checking in Deductive Databases:, Proc. Of the 13th VLDB, England, 1987, pp. 61-69

KUMAR87 Kumar, A.; Stonebraker, M.; (1987), "The effect of Join Selectiveties on optimal nesting order", SIGMOD Record, V16(1), p.p. 28-42

LAUSE97 Lausen, G.; Vossen, G.; "Models and Languages of Object-Oriented Databases"; Addison-Wesley, 1997, pp. 218

LELLA01 Lellahi, K.;. Zamulin, A.; "Object-Oriented Database as a Dynamic System With Implicit State", Proceed. 5th ADBIS Conference, Vilnus, Lithuania, 2001, LNCS 2151; pp. 239-252

LEUNG93 Leung, T., W.; Mitchell, G.; Subramania, B.; Vance, B.; Vanderberg, S., L.; Zdonik, S., B.; "The AQUA Data Model and Algebra"; Technical Report No. CS-93-09, Brown University; 1993.

LIEBE86 Lieberman, H.; "Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems", OOPSLA'86 Conf. Proceed., USA, ACM SIGPLAN N21(11),1986, pp.214-223

LISKO87 Liskov, B.; "Data Abstraction and Hierarchy", OOPSLA 1987 Addendum to the Proceedings, p.17-34

LISKO99 Liskov, B.; Wing, J.; "A behavioral notion of subtyping", ACM Transactions on Programming Languages and Systems (TOPLAS), V16/6, November 1994, pp. 1811 - 1841

LLOYD84 Lloyd, J, W; Topor, R, W; "Making logic more expressive", J. of Logic Programming, v1(3), 1984, pp. 225-240.

LLOYD85 Lloyd, J, W; Topor, R, W; "A Basis for Deductive Database Systems", J. of Logic Programming, v2(1), 1985, pp. 93-109.

MAEIR86 Maeir, D.; "A logic of objects", in Workshop on Foundations of Deductive and Logic Programming, Washington D.C., 1986, p.p. 6-26

MAEIR88 Maier, D.; Warren, D., S.; Computing with Logic, 1988,The Benjamin/Cummings Publishing Company, Inc

MAIER79 Maier, D; Mendelzon, A, O; Sagiv, Y; Testing implication of data dependencies, ACM Tran. On Database Systems, v4(4), 1979, pp. 455-469

MAIER83 Maier, D; The theory of relational databases, Pitman Pub. Ltd, 1983

MAKIN77 Makinouchi. A.; "A Consideration of Normal Form on Not-necessarily Normalized Relations in the Relational Data Model", Proceedings of the International Conference on Very Large Databases, 1977, pp. 447-453

MAYWO00 May, W.; "FLORiD User Manual", version 3.0, University of Freiburg, Gemany, 2000, pp. 36

MEIJE04 Meijer, E.; Drayton. P.; Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, In OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004.

MEYER96 Meyer, B.; "Object-Oriented Software Construction" ; Prentice Hall; second edition, 1996

MILNE90 Milner, R,; Tofte, M.; Harper, R.; "The Definition of Standard ML", MIT Press, 1990

MINKE87 Minker, J.; (Ed.); "Foundations of deductive databases and logic programming", Morgan Kaufmann Publ. Inc, 1987

MISHR84 Mishra, P.; "Towards a Theory of Types in Prolog", International Symposium on Logic Programming,: 1984, p.p. 289-298

MITCH03 Mitchell, J., C.; "Concepts in programming languages", Cambridge University Press, 2003

MITCH88 Mitchell, J., C.; Plotkin, G., D.; "Abstract Types Have Existential Type", ACM Transactions on Programming Languages and Systems, Vol. 10, No. 3, July 1988, pp. 470-502

NGPAU81 Ng, P.; "Further Analysis of the Entity-Relationship Approach to Database Design", Trans. Of Software Engineering, V7(1),1981, pp. 85-99

NICOL78 Nicolas, J, M; Yazdanian, K; 'Integrity checking in deductive databases', in Logic and Databases, Eds. Gallaire, H; and Minker, J; 1978, pp. 325-344

ODERS97 Odersky, M.; Wadler, P.; "Pizza into Java: Translating theory into practice",. Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997

ODERS10 Odersky, M.; Spoon, M.; Venners, B.; "Programming in Scala"; 2nd edition, artima; 2010, pp. 883

OMGRP06 Object Management Group; "Object Constraint Language OMG"; Available Specification Version 2.0, 2006

OMGRP08 Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 1: CORBA Interfaces, Object Management Group, 2008

OMGRP11 Object Management Group; "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)"; Available Specification Version 1.1; 2011

OUSTE98 Ousterhout. J., K.; Scripting: Higher-Level Programming for the 21st Century, Computer, 31(3), pp. 23-30, 1998

PALSB96 Palsberg, J.; Type Inference for Objects, Computing Surveys, v 28(2) , June 1996, pp. 358-359

PARNA72 Parnas, D.; On the criteria to be used in decomposing systems into modules, Communications of the ACM, v15,1972, pp. 1053-1058

PIERC92 Pierce, B, C; Bounded Quantification is un-decidable, 19th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, New Mexico, 1992, pp.305-315

PORTE92 Porter, H.,H.; "Separating the subtype hierarchy from the inheritance of implementation", Journal of Object-Oriented Programming, V(4).9, 1992, pp.20-29

ROBIN65 Robinson, J., A.; "A machine-oriented logic based on the resolution principle", Journal of the ACM, 12(1), 1965

ROTHM87 Roth, M.; Korth, H.; Batory, D.; "SQL/NF: A Query Language for 1NF Relational Databases"; Information. Systems, vo. 12(1), 1987; pp. 99-114.

ROTHM88 Roth, M., A.; Korth, H., F.; Silberschatz, A.; "Extended Algebra and Calculus for Nested Relational Databases"; ACM Trans. Database Syst.; vol. 13(4); 1988; pp.389-417

SADRI87 Sadri, F.; Kowalski, R.; "An application of general purpose theorem prover to database integrity", in Minker (ed) Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, 1987

SAVNI99 Savnik, I.; Tari, T.; Mohoric, T.; "QAL: A Query Algebra of Complex Objects"; Data & Knowledge Eng. Journal, vo. 30(1), 1999; pp.57-94.

SCHEK85 Schek, H., J.; "Towards a Basic Relational NF2 Algebra Processor"; Proceed. of the Int. Conf. FODO, Kyoto, Japan; 1985; pp.173-182

SCHEK86 Schek, H., J.; Scholl, M.; "The Relational Model with Relation-Valued Attributes"; Information Systems; vol. 11(2); 1986; pp. 137-147

SCHOO86 Scholl, M., H.; "Theoretical Foundations of Algebraic OptimizationUtilizing Unnormalized Relations"; Proceed. International Conference on Database Theory, Rome, Italy; 1986; pp. 380-396

SCHUS11 Schuster, W.; Interview with Rob Pike; "Rob Pike on Google Go: Concurrency, Type System, Memory Management and GC", InfoQ, on Feb 25, 2011 (http://www.infoq.com/interviews/pike-google-go)

SCOTT76 Scott, S., S.; "Data Types as Lattices"; SIAM J. Computer;. V 5(3), 1976, pp. 522-587

SELIN78 Selinger, P., G.; Astrahan, M., M.; Chamberlin, D., D.; Lorie, R., A.; Price, T., G.; "Access Path Selection in a Relational Database Management System", Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, 1979, pp. 23-34

SHAWS90 Shaw, G.; M.; Zdonik, S. B.; "A Query Algebra for Object Oriented Databases"; Proceed. of the 6th Intern. Conf. on Data Engineering, Los Angeles, USA; 1990

SILBE10 Silberschatz, A.; Korth, H.; Sudarshan, S.; "Database System Concepts"; McGraw-Hill; 2010; pp. 1376

SIMON95 Simons, A., J., H.; "A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Language"; Ph.D. thesis, Uni. Of Sheffield; 1995

SMITH77 Smith, J., M.; Smith, D., C., P.; "Database Abstractions: Aggregation and Generalization"; ACM Transactions on Database Systems; V 2(2);1977, pp. 105-133

SMITH77B Smith, J., M.; Smith, D., C., P.; "Database Abstractions: Aggregation"; Comm.s ACM; vo. 20, 1977; pp. 405-413.

SMITH95 Smith, R., B.; Ungar, D.; "Programming as an experience: the inspiration of Self", ECOOP'95 Conference Proceedings, Aarhus, Denmark, August, 1995

SNYDE86 Snyder, A.; "Encapsulation and Inheritance in Object-Oriented Programming Languages", ACM - OOPSLA'86 Proceed., pp.38-45

STEEL84 Steel, G., L.; "Common Lisp the language"; Digital Press; 1984, pp.465

STEFI86 Stefik, M.; Bobrow, D., G.; "Object-Oriented Programming: Themes and Variations", AI Magazine, January, 1986, pp.40-62

STONE90A Stonebraker, M.; Rowe, L.A.; Hirohama, M.; "The Implementation of POSTGRES", IEEE Transactions on Knowledge and Data Engineering, V(2)1, 1990, pp.125-42

STONE98 Stonebraker, M.; Hellerstein, J., M.; "What goes round comes around", in Stonebraker, M. and Hellerstein J., "Readings in Database Systems", Morgan-Kaufman Publishers, 1998

STRAC67 Strachey, C.; "Fundamental Concepts in Programming Languages", Lecture Notes for International Summer School in Computer Programming, Denmark, 1967

STRAU90 Straube, D.; Ozsu, M., T.; "Queries and query processing in object-oriented database systems"; ACM Trans. on Office Information Systems, vo. 8(4), 1990; pp. 387-430

STRAU90A Straube, D., D.; Ozsu, M., T.; Queries and Query Processing in Object-Oriented Database Systems, ACM Transactions on Information Systems, V8, N4, October 1990, p387-430

STROU92 Stroustrup, B.; "The C++ Programmin Language", Addison-Wesley Publishing Co., Second Edition, 1992

STROU94 Stroustrup, B.; "The Design and Evolution of C++"; Addison-Wesley; 1994, pp. 480

SUBIE98 Subieta, K.; Leszczytowski, J.; "A Critique of Object Algebras", Technical Report at the Ins. of Computer Science Polish Acad. Sci., Warszawa, Poland; 1998

TEORE05 Teorey, T., J.; Lightstone, S., S.; Nadeau, T.; "Database Modeling and Design: Logical Design"; Morgan Kaufmann, 4th Edition, 2005, pp. 296

THOMA86 Thomas, S., J.; Fischer P., C.; "Nested Relational Structures"; Advances in Computing Research; vol. 3; 1986; pp. 269-307

ULLMA87 Ullman, J., D.; "Database Theory: Past and Future", ACM PODS; 1987, pp. 1-10

ULLMA90 Ullman, J., D.; Principles of Database and Knowledge-Base Systems, Vol. I and II, W.H. Freeman & Co., 1990

VALDU87 Valduriez, P.; "Join Indices", ACM Tran. of Database Systems, V12(2), 1987, pp.218-246

VELLA00 Vella, J.; Musu, J.; "Materialised View Performance in a Data Warehouse Environment"; XXXIVth Intern. Conf. MOSIS/ISM 2000, Czech Republic, 2000; pp. 160-170

VELLA09 Vella, J.; "Converting EERM into ODMG's ODL Constructs" ; ICOODB Conference, Switzerland, 2009, tutorial;

VELLA97 Vella, J.; "Converting EERM into an Object-Oriented Data Model"; XXIIth Proceeding of Association of Simula Users, France, 1997; pp.1-10

VERHE82 Verheijen, G.; van Bekkum, J.; "NIAM: An Information Analysis Method", Information Systems Design Methodologies, 1982, pp.32-55

VIERA99 Viara, E.; Barillot, E.; Vaysseix, G.; (1999), "The EyeDB OODBMS", in Proceed. of the International Database Engineering and Applications Symposium 1999, p.p. 390-402

VOSSE91 Vossen, G.; "Data models, database languages and database management systems"; Addison-Wesley, 1991, pp. 541

WARNE03 Warmer, J., B.; Kleppe; A., G.; "The Object Constraint Language: Getting Your Models Ready for Mda"; Addison-Wesley Professional, 2003; p.p 206;

WEGNE89A Wegner, P.; "Learning the Language", Byte, March 1989, pp.245-253

WEGNE89B Wegner, P.; Zdonik, S.,B.; "Models of Inheritance", Proceed. of 2nd Workshop Databases Programming Languages, 1989, pp.248-255

YANGG03 Yang, G.; Kifer. M.; "Reasoning about anonymous resources and meta statements on the Semantic Web. Journal on Data Semantics"; LNCS vo, 2800, 2003; pp. 69-98;

YANGG08 Yang, G.; Kifer, M.; Wan, H.; Zhao, C.; "Flora-2: User's Manual", SRI International & DCS at State University of NY at Stony Brook; 2008, pp. 160;

ZAMUL02 Zamulin, A.; "An Object Algebra for the ODMG Standard", Proceed. 6th ADBIS Conference, Bratislava, Slovakia, 2002, LNCS 2435, pp. 291-304

ZANIO83 Zaniolo, C.; "The Database Language GEM"; SIGMOD Conference, 1983; pp. 207-218

Appendix – GraphVIZ/Dot Specification (afterScott)

```
/* FLOGIC SCHEMA FOR VISUALISATION */
// SCHEMA NAME & VERSION: AFTERSCOTT (ALPHA)
/* HEADER */
DIGRAPH FLOGICENCODE {
RANKSEP=1.25;
/* CLASSES */
COURSE[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
DEPT[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
LECTURER[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
PERSON[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
PROJECT[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
PTSTUDENT[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
STUDENT[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
STUDUNITGRADE[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
TELEPHONE[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
UNIT[SHAPE=BOX,HEIGHT=.75,WIDTH=1.5];
/* STRUCTURES - WEAK ENTITIES */
ADDRESS[SHAPE=BOX,HEIGHT=.75,WIDTH=1.4,PERIPHERIES=2];
JOB[SHAPE=BOX,HEIGHT=.75,WIDTH=1.4,PERIPHERIES=2];
UNITYEAR[SHAPE=BOX,HEIGHT=.75,WIDTH=1.4,PERIPHERIES=2];
/* ISA GRAPH */
ISAPERSON[SHAPE=CIRCLE,HEIGHT=.5, WIDTH=.5, FONTSIZE=12, FIXEDSIZE=TRUE,
LABEL="DISJOINT"];
PERSON-
>ISAPERSON[LABEL="PARTIAL",FONTSIZE=10,DIR="BOTH",ARROWHEAD=NONE,ARROWTAIL="VEE",COLOR="
GRAY:GRAY",WEIGHT=10];
ISAPERSON->LECTURER[LABEL="SEMANTIC",ARROWHEAD=NONE,COLOR="GRAY:GRAY",FONTSIZE=10];
ISAPERSON->STUDENT[LABEL="SEMANTIC",ARROWHEAD=NONE,COLOR="GRAY:GRAY",FONTSIZE=10];
ISASTUDENT[SHAPE=CIRCLE,HEIGHT=.5, WIDTH=.5, FONTSIZE=12, FIXEDSIZE=TRUE,
LABEL="DISJOINT"];
STUDENT-
>ISASTUDENT[LABEL="PARTIAL",FONTSIZE=10,DIR="BOTH",ARROWHEAD=NONE,ARROWTAIL="VEE",COLOR=
"GRAY:GRAY",WEIGHT=10];
ISASTUDENT->PTSTUDENT[LABEL="SEMANTIC",ARROWHEAD=NONE,COLOR="GRAY:GRAY",FONTSIZE=10];
/* ATTRIBUTES */
COURSECNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=CNAME,STYLE=BOLD];
COURSE->COURSECNAME[ARROWHEAD=NONE];
DEPTDNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=DNAME,STYLE=BOLD];
DEPT->DEPTDNAME[ARROWHEAD=NONE];
PERSONFNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=FNAME,STYLE=BOLD];
PERSON->PERSONFNAME[ARROWHEAD=NONE];
PERSONGENDER[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=GENDER];
PERSON->PERSONGENDER[ARROWHEAD=NONE];
PROJECTPNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=PNAME,STYLE=BOLD];
PROJECT->PROJECTPNAME[ARROWHEAD=NONE];
PTSTUDENTLOADRATIO[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=LOADRATIO];
PTSTUDENT->PTSTUDENTLOADRATIO[ARROWHEAD=NONE];
STUDENTSTAGE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=STAGE];
STUDENT->STUDENTSTAGE[ARROWHEAD=NONE];
STUDUNITGRADEGRADE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=GRADE];
STUDUNITGRADE->STUDUNITGRADEGRADE[ARROWHEAD=NONE];
TELEPHONETPNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=TPNAME,STYLE=BOLD];
TELEPHONE->TELEPHONETPNAME[ARROWHEAD=NONE];
TELEPHONETPSTATE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=TPSTATE];
TELEPHONE->TELEPHONETPSTATE[ARROWHEAD=NONE];
TELEPHONETPTYPE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=TPTYPE];
TELEPHONE->TELEPHONETPTYPE[ARROWHEAD=NONE];
UNITASSESSTYPE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=ASSESSTYPE];
UNIT->UNITASSESSTYPE[ARROWHEAD=NONE];
UNITCREDITS[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=CREDITS];
UNIT->UNITCREDITS[ARROWHEAD=NONE];
UNITUNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=UNAME,STYLE=BOLD];
UNIT->UNITUNAME[ARROWHEAD=NONE];
ADDRESSALINE1[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=ALINE1];
ADDRESS->ADDRESSALINE1[ARROWHEAD=NONE];
ADDRESSALINE2[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=ALINE2];
ADDRESS->ADDRESSALINE2[ARROWHEAD=NONE];
ADDRESSASERNO[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=ASERNO,STYLE=BOLD];
ADDRESS->ADDRESSASERNO[ARROWHEAD=NONE];
DEPTROLEFIRSTROLE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=FIRSTROLE];
DEPTROLE->DEPTROLEFIRSTROLE[ARROWHEAD=NONE];
DEPTROLEOTHERROLE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=OTHERROLE];
DEPTROLE->DEPTROLEOTHERROLE[ARROWHEAD=NONE];
JOBJBUDGET[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=JBUDGET];
JOB->JOBJBUDGET[ARROWHEAD=NONE];
JOBJNAME[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=JNAME,STYLE=BOLD];
JOB->JOBJNAME[ARROWHEAD=NONE];
UNITYEARUSEMESTER[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=USEMESTER];
```

```
UNITYEAR->UNITYEARUSEMESTER[ARROWHEAD=NONE];
UNITYEARUSERNO[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=USERNO,STYLE=BOLD];
UNITYEAR->UNITYEARUSERNO[ARROWHEAD=NONE];
UNITYEARUYEAR[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=UYEAR];
UNITYEAR->UNITYEARUYEAR[ARROWHEAD=NONE];
LECTURERDEGREES[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,PERIPHERIES=2,LABEL=DEGREES];
LECTURER->LECTURERDEGREES[ARROWHEAD=NONE];
/* COMPOSITE RELATIONSHIP */
DEPTROLE[SHAPE=ELLIPSE,WIDTH=.75,HEIGHT=.5,LABEL=DROLE,PERIPHERIES=2];
DEPT->DEPTROLE[ARROWHEAD=NONE];
/* RELATIONSHIPS */
COURSE->STUDENT[LABEL="ENROLLED ON",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
COURSE->DEPT[LABEL="SPONSOSER",DIR="BOTH",ARROWTAIL="CROWTEE",ARROWHEAD="CROWODOT"];
COURSE->UNIT[LABEL="REQUIRES",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="CROWODOT"];
DEPT->LECTURER[LABEL="EMPLOYS",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
LECTURER->UNITYEAR[LABEL="TEACH",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
LECTURER->PROJECT[LABEL="LEADS",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
LECTURER->UNIT[LABEL="COORDINATES",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
PERSON->TELEPHONE[LABEL="HASPHONE",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEEODOT"];
PERSON->PROJECT[LABEL="WORKSON",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="CROWODOT"];
PROJECT->PROJECT[LABEL="CONTINOF",DIR="BOTH",ARROWTAIL="TEEODOT",ARROWHEAD="CROWODOT"];
STUDENT->STUDUNITGRADE[LABEL="TOOK
UNIT",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
UNITYEAR->STUDUNITGRADE[LABEL="TOOK
STUDENT",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
/* WEAK RELATIONSHIPS */
DEPT->ADDRESS[LABEL="LOCATES AT",DIR="BOTH",ARROWTAIL="TEEODOT",ARROWHEAD="TEETEE"];
PROJECT->JOB[LABEL="HASJOBS",DIR="BOTH",ARROWTAIL="CROWODOT",ARROWHEAD="TEETEE"];
UNIT->UNITYEAR[LABEL="VERSION OF",DIR="BOTH",ARROWTAIL="TEEODOT",ARROWHEAD="TEETEE"];
}
```

Appendix - EyeDB ODL Specifications (afterScott)

```
// SCHEMA

// ENUMERATED TYPES (DOMAINS)
  ENUM ASSESSMENT {
    ASSIGN,
    EXAM,
    INTERVIEW,
    TEST
  };
  ENUM DEGREE {
    BAT,
    DOC,
    MASTER,
    POSTDOC
  };
  ENUM LOAD {
    FOURTH,
    HALF,
    THIRD,
    TWELFTH
  };
  ENUM SEX {
    FEMALE,
    MALE
  };
  ENUM UNITGRADE {
    A,
    ABS,
    B,
    C,
    D,
    F
  };

// FORWARD REFERENCES
CLASS COURSE;
CLASS DEPT;
CLASS LECTURER;
CLASS PERSON;
CLASS PROJECT;
CLASS PTSTUDENT;
CLASS STUDENT;
CLASS STUDUNITGRADE;
CLASS TELEPHONE;
CLASS UNIT;
CLASS ADDRESS;
```

```
CLASS DEPTROLE;
CLASS JOB;
CLASS UNITYEAR;

// STRUCTURES (WEAK ENTITIES)
CLASS ADDRESS {
  ATTRIBUTE STRING ALINE1;
  ATTRIBUTE STRING ALINE2;
  ATTRIBUTE INT ASERNO;
  // RC11 2
  RELATIONSHIP DEPT * ADEPT INVERSE DEPT::MAINOFFICE;
  // PRI KEY IS [ASERNO]
  CONSTRAINT<UNIQUE> ON ASERNO; INDEX ON ASERNO;
  CONSTRAINT<NOTNULL> ON ADEPT;
  CONSTRAINT<NOTNULL> ON ALINE1;
  CONSTRAINT<NOTNULL> ON ALINE2;
  CONSTRAINT<NOTNULL> ON ASERNO;
 } ;
CLASS DEPTROLE {
  ATTRIBUTE STRING FIRSTROLE;
  ATTRIBUTE STRING OTHERROLE;
  // PRI KEY IS []
  CONSTRAINT<NOTNULL> ON FIRSTROLE;
  CONSTRAINT<NOTNULL> ON OTHERROLE;
 } ;
CLASS JOB {
  ATTRIBUTE DOUBLE JBUDGET;
  ATTRIBUTE STRING JNAME;
  // RC12 2
  RELATIONSHIP PROJECT * JPROJ INVERSE PROJECT::JOBS;
  // PRI KEY IS [JNAME]
  CONSTRAINT<UNIQUE> ON JNAME; INDEX ON JNAME;
  CONSTRAINT<NOTNULL> ON JBUDGET;
  CONSTRAINT<NOTNULL> ON JNAME;
  CONSTRAINT<NOTNULL> ON JPROJ;
 } ;
CLASS UNITYEAR {
  ATTRIBUTE INT USEMESTER;
  ATTRIBUTE INT USERNO;
  ATTRIBUTE INT UYEAR;
  // RC10 2
  RELATIONSHIP LECTURER * ULECTURER INVERSE LECTURER::TEACHES;
  // RC14 2
  RELATIONSHIP UNIT * UUNIT INVERSE UNIT::UVERSION;
  // RC72 1
  RELATIONSHIP SET < STUDUNITGRADE * > USTUDGRADES  INVERSE STUDUNITGRADE::GRADEUNIT;
  // PRI KEY IS [USERNO]
  CONSTRAINT<UNIQUE> ON USERNO; INDEX ON USERNO;
```

```
  CONSTRAINT<NOTNULL> ON ULECTURER;
  CONSTRAINT<NOTNULL> ON USEMESTER;
  CONSTRAINT<NOTNULL> ON USERNO;
  CONSTRAINT<NOTNULL> ON UUNIT;
  CONSTRAINT<NOTNULL> ON UYEAR;
 } ;

// CLASSES
CLASS COURSE {
  ATTRIBUTE STRING CNAME;
  // RC8 1
  RELATIONSHIP SET < UNIT * > REQUIRE  INVERSE UNIT::APPLICABLETO;
  // RC6 1
  RELATIONSHIP SET < DEPT * > SPONSORER  INVERSE DEPT::SPONSORS;
  // RC13 1
  RELATIONSHIP SET < STUDENT * > STUDENTS  INVERSE STUDENT::ENROLON;
  // PRI KEY IS [CNAME]
  CONSTRAINT<UNIQUE> ON CNAME; INDEX ON CNAME;
  CONSTRAINT<NOTNULL> ON CNAME;
  CONSTRAINT<NOTNULL> ON SPONSORER;
 } ;
CLASS DEPT {
  ATTRIBUTE STRING DNAME;
  ATTRIBUTE SET < DEPTROLE * > DROLE;
  // RC11 1
  RELATIONSHIP ADDRESS * MAINOFFICE INVERSE ADDRESS::ADEPT;
  // RC6 2
  RELATIONSHIP SET < COURSE * > SPONSORS  INVERSE COURSE::SPONSORER;
  // RC5 1
  RELATIONSHIP SET < LECTURER * > STAFF  INVERSE LECTURER::WORKSAT;
  // PRI KEY IS [DNAME]
  CONSTRAINT<UNIQUE> ON DNAME; INDEX ON DNAME;
  CONSTRAINT<NOTNULL> ON DNAME;
 } ;
CLASS PERSON {
  ATTRIBUTE STRING FNAME;
  ATTRIBUTE SEX GENDER;
  // RC1 1
  RELATIONSHIP SET < TELEPHONE * > TELNO  INVERSE TELEPHONE::TPISOF;
  // RC2 1
  RELATIONSHIP SET < PROJECT * > WORKSON  INVERSE PROJECT::STAFF;
  // PRI KEY IS [FNAME]
  CONSTRAINT<UNIQUE> ON FNAME; INDEX ON FNAME;
  CONSTRAINT<NOTNULL> ON FNAME;
  CONSTRAINT<NOTNULL> ON GENDER;
 } ;
CLASS LECTURER EXTENDS PERSON {
  ATTRIBUTE SET <DEGREE> DEGREES;
```

```
  // RC5 2
  RELATIONSHIP DEPT * WORKSAT INVERSE DEPT::STAFF;
  // RC9 1
  RELATIONSHIP SET < UNIT * > COORD  INVERSE UNIT::COORDBY;
  // RC3 1
  RELATIONSHIP SET < PROJECT * > PROLEAD  INVERSE PROJECT::LEADER;
  // RC10 1
  RELATIONSHIP SET < UNITYEAR * > TEACHES  INVERSE UNITYEAR::ULECTURER;
  // PRI KEY IS []
  CONSTRAINT<NOTNULL> ON COORD;
  CONSTRAINT<NOTNULL> ON WORKSAT;
 } ;
CLASS PROJECT {
  ATTRIBUTE STRING PNAME;
  // RC4 1
  RELATIONSHIP PROJECT * CONTINOF INVERSE PROJECT::CARRIESONIN;
  // RC3 2
  RELATIONSHIP LECTURER * LEADER INVERSE LECTURER::PROLEAD;
  // RC4 2
  RELATIONSHIP SET < PROJECT * > CARRIESONIN  INVERSE PROJECT::CONTINOF;
  // RC12 1
  RELATIONSHIP SET < JOB * > JOBS  INVERSE JOB::JPROJ;
  // RC2 2
  RELATIONSHIP SET < PERSON * > STAFF  INVERSE PERSON::WORKSON;
  // PRI KEY IS [PNAME]
  CONSTRAINT<UNIQUE> ON PNAME; INDEX ON PNAME;
  CONSTRAINT<NOTNULL> ON LEADER;
  CONSTRAINT<NOTNULL> ON PNAME;
 } ;
CLASS STUDENT EXTENDS PERSON {
  ATTRIBUTE STRING STAGE;
  // RC13 2
  RELATIONSHIP COURSE * ENROLON INVERSE COURSE::STUDENTS;
  // RC71 1
  RELATIONSHIP SET < STUDUNITGRADE * > UNITGRADES  INVERSE STUDUNITGRADE::GRADESTUDENT;
  // PRI KEY IS []
  CONSTRAINT<NOTNULL> ON ENROLON;
 } ;
CLASS PTSTUDENT EXTENDS STUDENT {
  ATTRIBUTE LOAD LOADRATIO;
  // PRI KEY IS []
  CONSTRAINT<NOTNULL> ON LOADRATIO;
 } ;
CLASS STUDUNITGRADE {
  ATTRIBUTE UNITGRADE GRADE;
  // RC71 2
  RELATIONSHIP STUDENT * GRADESTUDENT INVERSE STUDENT::UNITGRADES;
  // RC72 2
```

```
  RELATIONSHIP UNITYEAR * GRADEUNIT INVERSE UNITYEAR::USTUDGRADES;
  // PRI KEY IS [GRADESTUDENT,GRADEUNIT]
  CONSTRAINT<UNIQUE> ON GRADESTUDENT; INDEX ON GRADESTUDENT;
  CONSTRAINT<NOTNULL> ON GRADE;
  CONSTRAINT<NOTNULL> ON GRADESTUDENT;
 } ;
CLASS TELEPHONE {
  ATTRIBUTE STRING TPNAME;
  ATTRIBUTE STRING TPSTATE;
  ATTRIBUTE STRING TPTYPE;
  // RC1 2
  RELATIONSHIP PERSON * TPISOF INVERSE PERSON::TELNO;
  // PRI KEY IS [TPNAME]
  CONSTRAINT<UNIQUE> ON TPNAME; INDEX ON TPNAME;
  CONSTRAINT<NOTNULL> ON TPNAME;
  CONSTRAINT<NOTNULL> ON TPTYPE;
 } ;
CLASS UNIT {
  ATTRIBUTE ASSESSMENT ASSESSTYPE;
  ATTRIBUTE INT CREDITS;
  ATTRIBUTE STRING UNAME;
  // RC9 2
  RELATIONSHIP LECTURER * COORDBY INVERSE LECTURER::COORD;
  // RC8 2
  RELATIONSHIP SET < COURSE * > APPLICABLETO  INVERSE COURSE::REQUIRE;
  // RC14 1
  RELATIONSHIP SET < UNITYEAR * > UVERSION  INVERSE UNITYEAR::UUNIT;
  // PRI KEY IS [UNAME]
  CONSTRAINT<UNIQUE> ON UNAME; INDEX ON UNAME;
  CONSTRAINT<NOTNULL> ON ASSESSTYPE;
  CONSTRAINT<NOTNULL> ON COORDBY;
  CONSTRAINT<NOTNULL> ON CREDITS;
  CONSTRAINT<NOTNULL> ON UNAME;
 }
```

```
;
```

Appendix - EyeDB processing of afterScott schema Specifications

```
JOSEPH@ASPIREONE:~/WORKBENCH/EYEDB/FUFU$ CP $HOME/DROPBOX/SCOTT_2.ODL .
JOSEPH@ASPIREONE:~/WORKBENCH/EYEDB/FUFU$ SUDO EYEDBADMIN DATABASE DELETE FUFU
[SUDO] PASSWORD FOR JOSEPH:
JOSEPH@ASPIREONE:~/WORKBENCH/EYEDB/FUFU$ SUDO EYEDBADMIN DATABASE CREATE FUFU
JOSEPH@ASPIREONE:~/WORKBENCH/EYEDB/FUFU$ SUDO EYEDBODL -D FUFU -U SCOTT_2.ODL
UPDATING 'SCOTT_2' SCHEMA IN DATABASE FUFU...
ADDING CLASS COURSE
ADDING CLASS DEPT
ADDING CLASS LECTURER
ADDING CLASS PERSON
ADDING CLASS PROJECT
ADDING CLASS PTSTUDENT
ADDING CLASS STUDENT
ADDING CLASS STUDUNITGRADE
ADDING CLASS TELEPHONE
ADDING CLASS UNIT
ADDING CLASS ADDRESS
ADDING CLASS DEPTROLE
ADDING CLASS JOB
ADDING CLASS UNITYEAR
ADDING ATTRIBUTE ADDRESS::ALINE1
ADDING ATTRIBUTE ADDRESS::ALINE2
ADDING ATTRIBUTE ADDRESS::ASERNO
ADDING ATTRIBUTE ADDRESS::ADEPT
ADDING ATTRIBUTE DEPTROLE::FIRSTROLE
ADDING ATTRIBUTE DEPTROLE::OTHERROLE
ADDING ATTRIBUTE JOB::JBUDGET
ADDING ATTRIBUTE JOB::JNAME
ADDING ATTRIBUTE JOB::JPROJ
ADDING ATTRIBUTE UNITYEAR::USEMESTER
ADDING ATTRIBUTE UNITYEAR::USERNO
ADDING ATTRIBUTE UNITYEAR::UYEAR
ADDING ATTRIBUTE UNITYEAR::ULECTURER
ADDING ATTRIBUTE UNITYEAR::UUNIT
ADDING ATTRIBUTE UNITYEAR::USTUDGRADES
ADDING ATTRIBUTE COURSE::CNAME
ADDING ATTRIBUTE COURSE::REQUIRE
ADDING ATTRIBUTE COURSE::SPONSORER
ADDING ATTRIBUTE COURSE::STUDENTS
ADDING ATTRIBUTE DEPT::DNAME
ADDING ATTRIBUTE DEPT::DROLE
ADDING ATTRIBUTE DEPT::MAINOFFICE
ADDING ATTRIBUTE DEPT::SPONSORS
ADDING ATTRIBUTE DEPT::STAFF
ADDING ATTRIBUTE PERSON::FNAME
ADDING ATTRIBUTE PERSON::GENDER
```

```
ADDING ATTRIBUTE PERSON::TELNO
ADDING ATTRIBUTE PERSON::WORKSON
ADDING ATTRIBUTE LECTURER::FNAME
ADDING ATTRIBUTE LECTURER::GENDER
ADDING ATTRIBUTE LECTURER::TELNO
ADDING ATTRIBUTE LECTURER::WORKSON
ADDING ATTRIBUTE LECTURER::DEGREES
ADDING ATTRIBUTE LECTURER::WORKSAT
ADDING ATTRIBUTE LECTURER::COORD
ADDING ATTRIBUTE LECTURER::PROLEAD
ADDING ATTRIBUTE LECTURER::TEACHES
ADDING ATTRIBUTE PROJECT::PNAME
ADDING ATTRIBUTE PROJECT::CONTINOF
ADDING ATTRIBUTE PROJECT::LEADER
ADDING ATTRIBUTE PROJECT::CARRIESONIN
ADDING ATTRIBUTE PROJECT::JOBS
ADDING ATTRIBUTE PROJECT::STAFF
ADDING ATTRIBUTE STUDENT::FNAME
ADDING ATTRIBUTE STUDENT::GENDER
ADDING ATTRIBUTE STUDENT::TELNO
ADDING ATTRIBUTE STUDENT::WORKSON
ADDING ATTRIBUTE STUDENT::STAGE
ADDING ATTRIBUTE STUDENT::ENROLON
ADDING ATTRIBUTE STUDENT::UNITGRADES
ADDING ATTRIBUTE PTSTUDENT::FNAME
ADDING ATTRIBUTE PTSTUDENT::GENDER
ADDING ATTRIBUTE PTSTUDENT::TELNO
ADDING ATTRIBUTE PTSTUDENT::WORKSON
ADDING ATTRIBUTE PTSTUDENT::STAGE
ADDING ATTRIBUTE PTSTUDENT::ENROLON
ADDING ATTRIBUTE PTSTUDENT::UNITGRADES
ADDING ATTRIBUTE PTSTUDENT::LOADRATIO
ADDING ATTRIBUTE STUDUNITGRADE::GRADE
ADDING ATTRIBUTE STUDUNITGRADE::GRADESTUDENT
ADDING ATTRIBUTE STUDUNITGRADE::GRADEUNIT
ADDING ATTRIBUTE TELEPHONE::TPNAME
ADDING ATTRIBUTE TELEPHONE::TPSTATE
ADDING ATTRIBUTE TELEPHONE::TPTYPE
ADDING ATTRIBUTE TELEPHONE::TPISOF
ADDING ATTRIBUTE UNIT::ASSESSTYPE
ADDING ATTRIBUTE UNIT::CREDITS
ADDING ATTRIBUTE UNIT::UNAME
ADDING ATTRIBUTE UNIT::COORDBY
ADDING ATTRIBUTE UNIT::APPLICABLETO
ADDING ATTRIBUTE UNIT::UVERSION
CREATING BTREEINDEX 'INDEX<TYPE = BTREE, PROPAGATE = ON> ON ADDRESS.ASERNO' ON CLASS
'ADDRESS'...
CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU
```

```
PED_BY_KEY = 0;", PROPAGATE = ON> ON JOB.JNAME' ON CLASS 'JOB'...

CREATING BTREEINDEX 'INDEX<TYPE = BTREE, PROPAGATE = ON> ON UNITYEAR.USERNO' ON CLASS
'UNITYEAR'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON COURSE.CNAME' ON CLASS 'COURSE'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON DEPT.DNAME' ON CLASS 'DEPT'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON PERSON.FNAME' ON CLASS 'PERSON'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON PROJECT.PNAME' ON CLASS 'PROJECT'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON STUDUNITGRADE.GRADESTUDENT' ON CLASS
'STUDUNITGRADE'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON TELEPHONE.TPNAME' ON CLASS 'TELEPHONE'...

CREATING HASHINDEX 'INDEX<TYPE = HASH, HINTS = "INITIAL_SIZE = 0; INITIAL_OBJECT_COUNT =
0; EXTEND_COEF = 0; SIZE_MAX = 0; DATA_GROU

PED_BY_KEY = 0;", PROPAGATE = ON> ON UNIT.UNAME' ON CLASS 'UNIT'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON ADDRESS.ASERNO' ON
CLASS 'ADDRESS'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON ADDRESS.ADEPT' ON
CLASS 'ADDRESS'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON ADDRESS.ALINE1' ON
CLASS 'ADDRESS'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON ADDRESS.ALINE2' ON
CLASS 'ADDRESS'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON ADDRESS.ASERNO' ON
CLASS 'ADDRESS'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON DEPTROLE.FIRSTROLE'
ON CLASS 'DEPTROLE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON DEPTROLE.OTHERROLE'
ON CLASS 'DEPTROLE'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON JOB.JNAME' ON CLASS
'JOB'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON JOB.JBUDGET' ON
CLASS 'JOB'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON JOB.JNAME' ON CLASS
'JOB'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON JOB.JPROJ' ON CLASS
'JOB'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON UNITYEAR.USERNO' ON
CLASS 'UNITYEAR'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNITYEAR.ULECTURER'
ON CLASS 'UNITYEAR'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNITYEAR.USEMESTER'
ON CLASS 'UNITYEAR'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNITYEAR.USERNO' ON
CLASS 'UNITYEAR'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNITYEAR.UUNIT' ON
CLASS 'UNITYEAR'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNITYEAR.UYEAR' ON
CLASS 'UNITYEAR'...
```

```
CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON COURSE.CNAME' ON CLASS
'COURSE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON COURSE.CNAME' ON
CLASS 'COURSE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON COURSE.SPONSORER' ON
CLASS 'COURSE'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON DEPT.DNAME' ON CLASS
'DEPT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON DEPT.DNAME' ON CLASS
'DEPT'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON PERSON.FNAME' ON CLASS
'PERSON'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON PERSON.FNAME' ON
CLASS 'PERSON'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON PERSON.GENDER' ON
CLASS 'PERSON'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON LECTURER.COORD' ON
CLASS 'LECTURER'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON LECTURER.WORKSAT' ON
CLASS 'LECTURER'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON PROJECT.PNAME' ON
CLASS 'PROJECT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON PROJECT.LEADER' ON
CLASS 'PROJECT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON PROJECT.PNAME' ON
CLASS 'PROJECT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON STUDENT.ENROLON' ON
CLASS 'STUDENT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON PTSTUDENT.LOADRATIO'
ON CLASS 'PTSTUDENT'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON
STUDUNITGRADE.GRADESTUDENT' ON CLASS 'STUDUNITGRADE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON STUDUNITGRADE.GRADE'
ON CLASS 'STUDUNITGRADE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON
STUDUNITGRADE.GRADESTUDENT' ON CLASS 'STUDUNITGRADE'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON TELEPHONE.TPNAME' ON
CLASS 'TELEPHONE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON TELEPHONE.TPNAME' ON
CLASS 'TELEPHONE'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON TELEPHONE.TPTYPE' ON
CLASS 'TELEPHONE'...

CREATING UNIQUE_CONSTRAINT 'CONSTRAINT<UNIQUE, PROPAGATE = ON> ON UNIT.UNAME' ON CLASS
'UNIT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNIT.ASSESSTYPE' ON
CLASS 'UNIT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNIT.COORDBY' ON
CLASS 'UNIT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNIT.CREDITS' ON
CLASS 'UNIT'...

CREATING NOTNULL_CONSTRAINT 'CONSTRAINT<NOTNULL, PROPAGATE = ON> ON UNIT.UNAME' ON CLASS
'UNIT'...

CREATING ONE-TO-ONE RELATIONSHIP ADDRESS::ADEPT <-> DEPT::MAINOFFICE

CREATING ONE-TO-MANY RELATIONSHIP JOB::JPROJ <-> PROJECT::JOBS

CREATING ONE-TO-MANY RELATIONSHIP UNITYEAR::ULECTURER <-> LECTURER::TEACHES

CREATING ONE-TO-MANY RELATIONSHIP UNITYEAR::UUNIT <-> UNIT::UVERSION

CREATING MANY-TO-ONE RELATIONSHIP UNITYEAR::USTUDGRADES <-> STUDUNITGRADE::GRADEUNIT

CREATING MANY-TO-MANY RELATIONSHIP COURSE::REQUIRE <-> UNIT::APPLICABLETO

CREATING MANY-TO-MANY RELATIONSHIP COURSE::SPONSORER <-> DEPT::SPONSORS
```

```
CREATING MANY-TO-ONE RELATIONSHIP COURSE::STUDENTS <-> STUDENT::ENROLON
CREATING ONE-TO-ONE RELATIONSHIP DEPT::MAINOFFICE <-> ADDRESS::ADEPT
CREATING MANY-TO-MANY RELATIONSHIP DEPT::SPONSORS <-> COURSE::SPONSORER
CREATING MANY-TO-ONE RELATIONSHIP DEPT::STAFF <-> LECTURER::WORKSAT
CREATING MANY-TO-ONE RELATIONSHIP PERSON::TELNO <-> TELEPHONE::TPISOF
CREATING MANY-TO-MANY RELATIONSHIP PERSON::WORKSON <-> PROJECT::STAFF
CREATING MANY-TO-ONE RELATIONSHIP PERSON::TELNO <-> TELEPHONE::TPISOF
CREATING MANY-TO-MANY RELATIONSHIP PERSON::WORKSON <-> PROJECT::STAFF
CREATING ONE-TO-MANY RELATIONSHIP LECTURER::WORKSAT <-> DEPT::STAFF
CREATING MANY-TO-ONE RELATIONSHIP LECTURER::COORD <-> UNIT::COORDBY
CREATING MANY-TO-ONE RELATIONSHIP LECTURER::PROLEAD <-> PROJECT::LEADER
CREATING MANY-TO-ONE RELATIONSHIP LECTURER::TEACHES <-> UNITYEAR::ULECTURER
CREATING ONE-TO-MANY RELATIONSHIP PROJECT::CONTINOF <-> PROJECT::CARRIESONIN
CREATING ONE-TO-MANY RELATIONSHIP PROJECT::LEADER <-> LECTURER::PROLEAD
CREATING MANY-TO-ONE RELATIONSHIP PROJECT::CARRIESONIN <-> PROJECT::CONTINOF
CREATING MANY-TO-ONE RELATIONSHIP PROJECT::JOBS <-> JOB::JPROJ
CREATING MANY-TO-MANY RELATIONSHIP PROJECT::STAFF <-> PERSON::WORKSON
CREATING MANY-TO-ONE RELATIONSHIP PERSON::TELNO <-> TELEPHONE::TPISOF
CREATING MANY-TO-MANY RELATIONSHIP PERSON::WORKSON <-> PROJECT::STAFF
CREATING ONE-TO-MANY RELATIONSHIP STUDENT::ENROLON <-> COURSE::STUDENTS
CREATING MANY-TO-ONE RELATIONSHIP STUDENT::UNITGRADES <-> STUDUNITGRADE::GRADESTUDENT
CREATING MANY-TO-ONE RELATIONSHIP PERSON::TELNO <-> TELEPHONE::TPISOF
CREATING MANY-TO-MANY RELATIONSHIP PERSON::WORKSON <-> PROJECT::STAFF
CREATING ONE-TO-MANY RELATIONSHIP STUDENT::ENROLON <-> COURSE::STUDENTS
CREATING MANY-TO-ONE RELATIONSHIP STUDENT::UNITGRADES <-> STUDUNITGRADE::GRADESTUDENT
CREATING ONE-TO-MANY RELATIONSHIP STUDUNITGRADE::GRADESTUDENT <-> STUDENT::UNITGRADES
CREATING ONE-TO-MANY RELATIONSHIP STUDUNITGRADE::GRADEUNIT <-> UNITYEAR::USTUDGRADES
CREATING ONE-TO-MANY RELATIONSHIP TELEPHONE::TPISOF <-> PERSON::TELNO
CREATING ONE-TO-MANY RELATIONSHIP UNIT::COORDBY <-> LECTURER::COORD
CREATING MANY-TO-MANY RELATIONSHIP UNIT::APPLICABLETO <-> COURSE::REQUIRE
CREATING MANY-TO-ONE RELATIONSHIP UNIT::UVERSION <-> UNITYEAR::UUNIT

DONE
```

Appendix - ODMG Data Dictionary

The application program data requirements in a database environment are recorded in the schema definition, we have seen in the first chapter. The ODMG calls this the *metadata* and it is stored in the ODL Schema Repository. The ODMG metadata is a superset of ODM's IDL Interface Repository because of the richer "structures" (e.g. relationships) found in it. The standard specifies the interface of each constituent part of the metadata and provides a terse textual description of the properties and methods. The exact semantics of the operations are missing.

There are thirty-two interfaces with thirty inheritance relationships between them. Most of these inheritance relationships are of the single inheritance type but four relay on multiple inheritance. The longest depth of the *ISA* relationship is of four (e.g. **DICTIONARY** / **COLLECTION** / **TYPE** / **METAOBJECT** / **REPOSITORYOBJECT**).

A focal point is the **METAOBJECT** interface, and it has two descriptive attributes - **NAME** and **COMMENT**. The meta-object emanates a number of other specialised interfaces; namely **EXCEPTION**, **CONSTANT**, **PROPERTY** (that itself emanate the interfaces **ATTRIBUTE** and **RELATIONSHIP**), and **TYPE**. The **TYPE** interface is specialised into three other interfaces; namely **TYPEDEFINITION**, **PRIMITIVETYPE**, and **COLLECTION**.

In the case of **EXCEPTION**s, the interface specifies a N-M relationship with **OPERATION**s that can actually raise them (i.e. note that although relationships do flag exceptions these are really encoded into the 'form' and 'to' methods), and a 1-1 relationship to a **STRUCTURE** data type definition for the information being passed out by the exception handler.

The interface **CONSTANT** provides a look-up structure for statically associating values with names. The value is either a literal, or a reference to another constant or a constant expression and in either case we relate to the respective interface (i.e. **OPERAND**, **CONSTOPERAND**, and **ENUMERATION** respectively). Each constant has a **TYPE** and a **VALUE**.

The **PROPERTY** interface is divided (in totality) into the **ATTRIBUTE** and **RELATIONSHIP** interfaces. Each property instance is related to a data **TYPE** in a M-1 relationship. As some attribute instances are read-only this is shown though a Boolean method called

IS_READ_ONLY. In the case of relationship instances we need to keep details on the role of the relationship (i.e. **GET_CARDINALITY**) and the back traversal path (this is a 1-1 recursive relationship called **TRAVERSAL**).

The data **TYPE** interface holds information on data types and a myriad range of 1-N relationships to interfaces that require a data type quantification (i.e. **COLLECTION**, **DICTIONARY**, **SPECIFIER**, **UNION**, **OPERATION**, **PROPERTY**, **CONSTANT** and **TYPEDEFINITION**). Because the standard allows the introduction of new names for defined data types (i.e. instances of **TYPE**) then a specialised interface of type called **TYPEDEFINITION** is introduced. The **TYPEDEFINITION** interface has a N-1 relationship with the **TYPE** interface. Another specialisation of the data type interface is the **PRIMITIVETYPE** interface. This interface adds a read-only attribute called **PRIMITIVE_KIND** whose value is to be chosen from an enumerated list of primitive types recognised by the standard (e.g. Boolean, char, date, etc). An important specialisation of data type is the **COLLECTION** interface. The collection interface has a read-only attribute (i.e. **COLLECTIONKIND**) that takes a value from a list of collection types allowed by the standard (e.g. list, array, bag, set, etc), and an association to a data type to represented the collection item's data type. Two methods are specified to cater for the sizing and ordering property of the collection. The **DICTIONARY** interface is a specialisation of the collection interface.

We have seen that ODMG allows us to "scope" our definition of objects and names. To record these themes in the data dictionary the **SCOPE** interface is introduced. The operation **BIND** takes a **METAOBJECT** instance, and a name to establish the given object scope in the repository. A sub-class of scope is **DEFININGSCOPE**. This has an important 1-N relationship with **METAOBJECTS**, called **DEFINES**, that enumerates which objects are in the same scope. There is also an array of methods to create, add and remove different **REPOSITORYOBJECT** (more on this interface later). Examples of such methods include **CREATE_COLLECTION**, **ADD_UNION**, and **REMOVE_CONSTANT**.

We have already stated that some interfaces have a multiple inheritance and one example is the **OPERATION** interface that inherits from **SCOPE** and **METAOBJECT**. An operation is

(like an interface's attribute) a **METAOBJECT** but due to the affects of dynamic dispatching require an operation to have a scope (thus the need to inherit from **SCOPE**). An operation instance needs to hold information about the data type of its parameters, the data type of its result and the list of exceptions that can be raised by the operation instance. This information is recorded through three relationships called **SIGNATURE** (to interface **PARAMETER** in a 1-N), **RESULT** (to interface **TYPE** in a N-1), and **EXCEPTIONS** (to interface **EXCEPTION** in a N-M).

Other important **METAOBJECT**s are the **INTERFACE** (itself) and the **CLASS**. The **CLASS** inherits from the **INTERFACE** while the **INTERFACE** inherits (through multiple inheritance mechanism) from **TYPE** and **DEFININGSCOPE**. The interface meta description (i.e. its interface!) include a recursive N-M relationship to depict the inheritance graph (this is implemented through the two relationships **INHERITS** and **DERIVES**). Also a number of methods are available to add and remove attributes, relationships and methods from an interface. These are called the **ADD_ATTRIBUTE**, **ADD_RELATIONSHIP** and **ADD_OPERATION** methods (and their 'remove' counterpart). In the **CLASS** interface we find a recursive N-1 relationship to implement the class extends graph (through the relationships **EXTENDER** and **EXTENSIONS**) and attributes to record the extents and keys associated with the class instances. In the case of the keys attribute it should really be a relationship to the attribute interface!

The ODMG's object model allows the designer to introduce new data types composed from other data types. The generic interface for these is the **SCOPEDTYPE** and it inherits from **TYPE** and **SCOPE**. Three possible **SCOPEDTYPE**s are catered here and these are the data structures, the enumerated types and the union data type (through the **STRUCTURE**, **ENUMERATION**, and **UNION TYPE**). The speciality of **STRUCTURE** is the list of typed attributes that compose it. The speciality of the **ENUMERATION** interface is a list of constants that comprise the particular instance (through the 1-N **ELEMENTS** relationship to the **CONSTANT** interface).

A module, and its specialisation repository, is a collection of meta-objects within a context of a defining scope. Note that the data dictionary is an instance of the repository interface.

This interface has a number of important methods for creating and removing modules (themselves), interfaces and classes.

In some contexts a data type requires naming, for example the name of an argument in a method. A generic class is the **SPECIFIER**, a subclass of **REPOSITORYOBJECT**, which has the following sub-classes; **MEMBER**, **UNIONCASE** and **PARAMETER**. Each specifier has a N-1 relationship to the **TYPE** interface. Interesting to note is the case of **PARAMETER**s where we need to annotate each argument with its argument-passing mode (i.e. IN, OUT or INOUT).

The **REPOSITORYOBJECT** interface has another ramification starting with the **OPERAND** interface. **OPERAND**s are an interface whose instances would be all the constant values in the object collection. These constant values can be literals (i.e. **LITERAL** interface inherits from operand), constant operands (i.e. **CONSTOPERAND** interface inherits form operand), and constant expressions (i.e. **EXPRESSION** interface inherits from operand).

One of the first interfaces described was the **METAOBJECT** (which inherits from **REPOSITORYOBJECT** interface); yet the description was incomplete. In fact each **METAOBJECT** instance is related to a **DEFININGSCOPE** instance through the **DEFINEDLN** N-1 relationship.

What's Missing From ODG's Meta Data

The most obvious missing "item" is detail; for example many of the methods mentioned have a short textual phrase for each and its signature.

Other significant items found in a data dictionary but not present in ODMG's proposal include: logical views, users, security profiles, and data constraints. We iterate that this is a significant shortfall that cannot be attributed to implementation dependency.